

DEITEL® DEVELOPER SERIES

Modern C++

Functional-Style Programming

Concepts

Templates

Copy/Move Semantics

Spaceship Operator

Smart Pointers

Text Formatting

Modules

Security

Standard Library

C++20

для программистов

Метод готовых объектов

Executors

Design Patterns

Coroutines

Ranges/Views

Performance

Contracts

Open Source Libraries

jthread

Parallel Algorithms

Concurrency

Lambdas

ПОЛ ДЕЙТЕЛ • ХАРВИ ДЕЙТЕЛ



C++20 для программистов

Пол Дейтел, Харви Дейтел

Часть 1
Базовые возможности C++20: быстрый старт. Процедурное программирование

1. Введение и тест-драйв популярных бесплатных компиляторов C++
Краткая справка по ООП. C++ в Windows®, MacOS® и Linux (Visual C++®, g++, Xcode® и clang++)

2. Азы программирования на C++
Базовые понятия C++. Введение в метод готовых объектов (МГО): быстрое создание объектно-ориентированных программ с помощью библиотек. МГО: работа с объектами класса string

3. Управляющие инструкции (часть 1)
Основы форматирования текста в C++20. МГО: операции с чрезвычайно большими целыми числами с помощью бесплатных библиотек с открытым кодом

4. Управляющие инструкции (часть 2)
МГО: библиотека miniz-cpp для записи и чтения ZIP-файлов

5. Функции и шаблоны функций
МГО: *Lnfylun Lhqtomh Wjtz Qarcv: Qjwazkrplm xzz Xndmwwqhlz* (зашифрованная фраза из нашего примера шифрования с закрытым ключом)

- Язык C++ как средство создания критически важных систем.
- «Большая четверка» C++20: диапазоны, концепты, модули и корутины.
- Сотни примеров исходного кода с комментариями.
- Адрес для общения с авторами: `deitel@deitel.com`

Часть 2
Контейнеры, диапазоны C++20, указатели, строки и файлы

6. Массивы, векторы, библиотека ranges и функциональное программирование
Основы функционального программирования. МГО: шаблон класса vector

7. Указатели в современном C++ (фактор риска)
Проблемы указателей и безопасные альтернативы. МГО: класс span в C++20

8. Строки и их представления, текстовые файлы, CSV-файлы и регулярные выражения
МГО: чтение и анализ набора данных «Титаник» в формате CSV



- Инструменты статического анализа кода.
- Загрузка исходного кода по адресу: `https://deitel.com/cpp20fp`

Часть 3
Современное объектно-ориентированное программирование и исключения

9. Пользовательские классы
МГО: сериализация объектов в формате JSON

10. ООП: наследование и динамический полиморфизм
Наследование интерфейса, идиома невиртуального интерфейса (NVI), std::variant и std::visit

11. Перегрузка операторов, семантика копирования и перемещения, умные указатели и RAII
Создание уникальных классов: пользовательский класс MyArray. Оператор трехстороннего сравнения (<=>) в C++20, управление ресурсами с помощью RAII

12. Исключения и обзор контрактного программирования

Часть 4
Обобщенное программирование: шаблоны, концепты и метапрограммирование

13. Контейнеры и итераторы стандартной библиотеки
Работа с готовыми структурами данных

14. Алгоритмы стандартной библиотеки, диапазоны и представления C++20
Функциональное программирование

15. Шаблоны, концепты (C++20) и метапрограммирование
Динамический полиморфизм, шаблоны функций, сокращенные шаблоны функций в C++20, шаблоны классов, вариативные шаблоны, выражения свертки

Часть 5
Более сложные темы: модули, параллельные алгоритмы, конкурентность и корутины

16. Модули C++20: технология разработки больших программ
Импорт заголовков, заголовочные единицы, объявление модуля, фрагменты модуля, разделы

17. Параллельные алгоритмы и конкурентность: высокоуровневый подход
Программирование для многоядерных процессоров с параллельными алгоритмами C++17. Конкурентность, многопоточность

18. Корутины (C++20)
co_await, co_yield, co_return. Библиотеки поддержки корутин, генераторы, исполнители и задачи

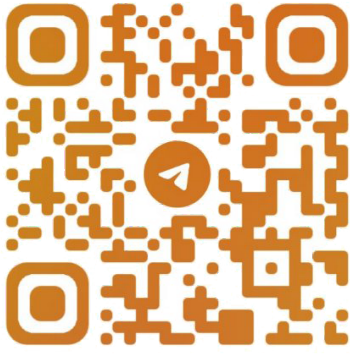
Часть 6
Онлайн-главы (на английском языке)

19. (Online) Stream I/O and C++20 Text Formatting

20. (Online) Other Topics and a Look Toward C++23

- Ресурсы для разработчиков: GitHub®, StackOverflow®, проекты с открытым исходным кодом и многие другие.
- Контейнеры Docker с компиляторами GNU g++ и Clang.
- Обзор ожидаемых возможностей C++23.
- Блог: `https://deitel.com/blog`





@CODELIBRARY_IT

ПИТЕР®



DEITEL® DEVELOPER SERIES

C++20 for Programmers

An Objects-Natural Approach

Paul Deitel • Harvey Deitel



Pearson

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo



DEITEL® DEVELOPER SERIES

C++20 для программистов

Метод готовых объектов

Пол Дейтел • Харви Дейтел

ПИТЕР

Санкт-Петербург • Москва • Минск

2024

ББК 32.973.2-018.1
УДК 004.43
Д27

Дейтел Пол, Дейтел Харви

Д27 C++20 для программистов. — СПб.: Питер, 2024. — 1056 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-2359-9

Программисты с опытом работы на других языках высокого уровня смогут на практике освоить современный C++ и «большую четверку» его новых возможностей: диапазоны, концепты, модули и корутины.

200+ практических примеров реального исходного кода позволят быстро овладеть идиомами современного C++, используя популярные компиляторы: Visual C++®, GNU® g++, Apple® Xcode® и LLVM®/Clang. Знание базы позволит перейти к контейнерам стандартной библиотеки C++ array и vector; функциональному программированию с диапазонами и представлениями C++20; строкам, файлам и регулярным выражениям; объектно-ориентированному программированию с классами, наследованием, динамическим и статическим полиморфизмом; перегрузке операторов, семантике копирования и перемещения, RAII и умным указателям; исключениям и ожидаемым в C++23 контрактам; контейнерам, итераторам и алгоритмам стандартной библиотеки; шаблонам, концептам C++20 и метапрограммированию; модулям C++ 20 и технологии разработки больших программ; конкурентности, параллелизму, параллельным алгоритмам стандартной библиотеки C++17 и C++20 и корутинам C++20.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1
УДК 004.43

Права на издание получены по соглашению с Pearson Education Inc. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-0136905691 англ.

Authorized translation from the English language edition, entitled C++20 FOR PROGRAMMERS, 3rd Edition by PAUL DEITEL; HARVEY DEITEL, published by Pearson Education, Inc. publishing as Pearson, © 2022

ISBN 978-5-4461-2359-9

© Перевод на русский язык ООО «Прогресс книга», 2024

© Издание на русском языке, оформление ООО «Прогресс книга», 2024

© Серия «Библиотека программиста», 2024

Краткое содержание

Предисловие 24

Об авторах 46

От издательства 47

Перед началом работы 48

Глава 1. Введение и тест-драйв популярных бесплатных компиляторов C++ 56

Глава 2. Азы программирования на C++ 79

Глава 3. Управляющие инструкции (часть 1) 101

Глава 4. Управляющие инструкции (часть 2) 137

Глава 5. Функции и шаблоны функций 175

Глава 6. Массивы, векторы, библиотека ranges и функциональное программирование 235

Глава 7. Указатели в современном C++ (фактор риска) 277

Глава 8. Строки и их представления, текстовые файлы, CSV-файлы и регулярные выражения 315

Глава 9. Пользовательские классы 370

Глава 10. ООП: наследование и динамический полиморфизм 446

Глава 11. Перегрузка операторов, семантика копирования и перемещения, умные указатели 535

Глава 12. Исключения и обзор контрактного программирования 597

Глава 13. Контейнеры и итераторы стандартной библиотеки 642

Глава 14. Алгоритмы стандартной библиотеки, диапазоны и представления C++20 701

Глава 15. Шаблоны, концепты (C++20) и метапрограммирование 779

Глава 16. Модули (C++20): технология разработки больших программ 875

Глава 17. Параллельные алгоритмы и конкурентность: высокоуровневый подход 931

Глава 18. Корутины (C++20) 1024

Приложение А. Приоритеты и группировка операторов 1052

Приложение Б. Набор символов 1055

Оглавление

Предисловие **24**
Современный C++ 24
В курсе событий 25
Кому пригодится эта книга 25
Готовые файлы с исходным кодом программ 25
Три профессиональных компилятора на выбор 26
Учебный метод готовых объектов 26
Что такое готовые объекты? 27
Бесплатные классы 27
Проект Boost 27
Примеры кода с готовыми объектами 28
Краткое содержание книги 28
Часть 1. Быстрый старт и базовые возможности C++ 29
Часть 2. Массивы, указатели и строки 30
Часть 3. Объектно-ориентированное программирование 31
Часть 4. Контейнеры, итераторы и алгоритмы стандартной библиотеки 33
Часть 5. Более сложные темы 34
Приложения 35
Дополнительные материалы на сайте deitel.com 35
C++ Core Guidelines 36
Библиотека Guidelines Support Library (GSL) 36
Профессиональные статические анализаторы кода 37
Интернет-ресурсы для программистов 37
Stack Overflow 37
GitHub 38
Docker 39
Некоторые ключевые документы и ресурсы по C++ 39
Документация 40
Блоги 40
Дополнительные ресурсы 41
Сайты, где ответят на ваши вопросы 41
Общение с авторами 41

Благодарности ... 41
Рецензенты ... 42
Артур О'Двайр ... 43
GitHub ... 43
Мэтт Годболт и Compiler Explorer ... 43
Дитмар Куль ... 44
Райнер Гримм ... 44
Брайан Гетц ... 44
Разработчики программ с открытым исходным кодом и блогеры ... 44
Поисковая система Google ... 45
Grammarly ... 45

Об авторах ... 46

От издательства ... 47

Перед началом работы ... 48
Условные обозначения ... 48
Загрузка примеров кода ... 48
Компиляторы, на которых мы тестировали исходный код ... 49
Не все примеры кода совместимы со всеми тремя компиляторами ... 49
Установка Visual Studio Community Edition в Windows ... 50
Установка Xcode в macOS ... 50
Установка актуальной версии GNU C++ ... 50
Установка GNU Compiler Collection в Ubuntu Linux, запущенной в подсистеме Windows для Linux ... 51
Docker и контейнеры Docker ... 52
Установка Docker ... 52
Загрузка контейнера Docker, содержащего GNU Compiler Collection ... 52
Загрузка контейнера Docker, содержащего Clang ... 53
Ответы на ваши вопросы по C++ ... 53
Онлайн-документация по C++ ... 54
О библиотеке форматирования текста {fmt} ... 54
Средства статического анализа кода ... 54

Глава 1. Введение и тест-драйв популярных бесплатных компиляторов C++ ... 56
1.1. Введение ... 57
1.2. Тест-драйв: компиляция приложения, написанного на C++20 ... 58
1.2.1. Компиляция и запуск приложения в Windows с помощью Visual Studio 2022 ... 59

1.2.2. Компиляция и запуск приложения в macOS с помощью Xcode 63
1.2.3. Компиляция и запуск приложения в Linux с помощью g++ 67
1.2.4. Компиляция и запуск приложения в контейнере Docker с помощью g++ 70
1.2.5. Компиляция и запуск приложения в контейнере Docker с помощью clang++ 71
1.3. Закон Мура, многоядерные процессоры и многопоточное программирование 73
1.4. Что такое объектно-ориентированное программирование? 75
1.5. Итоги 78

Глава 2. Азы программирования на C++ 79
2.1. Введение 80
2.2. Первая программа на C++: вывод текста на экран 80
2.3. Модификация нашей первой программы на C++ 85
2.4. Вторая программа на C++: сложение целых чисел 86
2.5. Арифметические операторы 90
2.6. Принятие решений: операторы равенства и сравнения 92
2.7. Готовые объекты: создание и использование объектов класса string стандартной библиотеки 96
2.8. Итоги 100

Глава 3. Управляющие инструкции (часть 1) 101
3.1. Введение 102
3.2. Управляющие структуры 102
3.2.1. Последовательность 103
3.2.2. Инструкции выбора 104
3.2.3. Инструкции цикла 104
3.2.4. Комбинации управляющих инструкций 105
3.3. Инструкция одиночного выбора if 106
3.4. Инструкция двойного выбора if...else 107
3.4.1. Вложенные инструкции if...else 108
3.4.2. Блоки 109
3.4.3. Тернарный условный оператор ?: 110
3.5. Инструкция цикла while 110
3.6. Цикл со счетчиком 111
3.6.1. Реализация цикла со счетчиком 112
3.6.2. Целочисленное деление с усечением 113
3.7. Цикл с контрольным значением 114
3.7.1. Реализация цикла с контрольным значением 114
3.7.2. Явное и неявное преобразование основных типов данных 117
3.7.3. Форматирование чисел с плавающей точкой 118

3.8. Вложенные управляющие инструкции ... 119
3.8.1. Постановка задачи ... 119
3.8.2. Реализация программы ... 120
3.8.3. Предотвращение сужающих преобразований с помощью скобочной инициализации ... 122
3.9. Составные операторы присваивания ... 123
3.10. Операторы инкремента и декремента ... 123
3.11. Несовместимость основных типов данных на разных платформах ... 127
3.12. Готовые объекты: работа с целыми числами произвольной величины ... 127
3.13. Форматирование текста функцией format ... 133
3.14. Итоги ... 135

Глава 4. Управляющие инструкции (часть 2) ... 137
4.1. Введение ... 138
4.2. Как работает цикл со счетчиком ... 138
4.3. Инструкция цикла for ... 139
4.4. Инкремент и декремент счетчика в цикле for ... 142
4.5. Программа для вычисления суммы четных чисел ... 143
4.6. Программа для расчета сложных процентов ... 144
4.7. Инструкция цикла do...while ... 148
4.8. Инструкция множественного выбора switch ... 150
4.9. Инструкции выбора с инициализаторами ... 156
4.10. Инструкции break и continue ... 157
4.11. Логические операторы ... 159
4.11.1. Оператор && (логическое И) ... 159
4.11.2. Оператор || (логическое ИЛИ) ... 160
4.11.3. Вычисление по короткой схеме ... 161
4.11.4. Оператор ! (логическое НЕ) ... 162
4.11.5. Программное создание таблиц истинности для логических операторов ... 162
4.12. Типичные ошибки при работе с операторами равенства (==) и присваивания (=) ... 164
4.13. Готовые объекты: создание и чтение ZIP-файлов ... 166
4.14. Форматирование текста с указанием ширины полей и точности ... 171
4.15. Итоги ... 174

Глава 5. Функции и шаблоны функций ... 175
5.1. Введение ... 176
5.2. Функции и классы — основные компоненты программ ... 176
5.3. Функции математической библиотеки ... 177
5.4. Определения функций и прототипы функций ... 180

5.5. Порядок вычисления аргументов функции ... 183
5.6. Дополнительные сведения о прототипах функций и принудительном приведении аргументов ... 184
5.6.1. Сигнатуры функций и прототипы функций ... 184
5.6.2. Принудительное приведение аргументов ... 184
5.6.3. Правила продвижения аргументов и неявные преобразования ... 185
5.7. Заголовки стандартной библиотеки C++ ... 187
5.8. Генерация случайных чисел ... 190
5.8.1. Бросаем игровой кубик ... 191
5.8.2. Бросаем кубик 60 миллионов раз ... 192
5.8.3. Инициализация генератора случайных чисел ... 194
5.8.4. Инициализация генератора случайных чисел с помощью random_device ... 195
5.9. Программа «Игра в кости» и знакомство с перечислениями ... 196
5.10. Правила областей видимости ... 202
5.11. Встроенные функции ... 207
5.12. Ссылки и ссылочные параметры ... 208
5.13. Аргументы по умолчанию ... 212
5.14. Оператор разрешения области видимости ... 214
5.15. Перегрузка функций ... 214
5.16. Шаблоны функций ... 218
5.17. Рекурсия ... 221
5.18. Пример рекурсивных вычислений: числа Фибоначчи ... 224
5.19. Рекурсия или итерация? ... 228
5.20. Lnfylun Lhqtomh Wjtz Qarcv: Qjwazkrplm xzz Xndmwwqhlz ... 230
5.21. Итоги ... 234

Глава 6. Массивы, векторы, библиотека ranges и функциональное программирование ... 235
6.1. Введение ... 236
6.2. Массивы ... 237
6.3. Объявление массива ... 237
6.4. Инициализация массива в цикле ... 238
6.5. Инициализация массива со списком инициализаторов ... 241
6.6. Цикл for для диапазона значений (C++11) и цикл for для диапазона значений с инициализатором (C++20) ... 242
6.7. Заполнение массива и спецификатор constexpr ... 245
6.8. Суммирование элементов массива ... 247
6.9. Визуализация данных массива на простой диаграмме ... 248
6.10. Использование элементов массива в качестве счетчиков ... 250

6.11. Использование массива для подсчета результатов опроса ... 251
6.12. Сортировка массива и поиск в массиве ... 252
6.13. Многомерные массивы ... 255
6.14. Основы функционального программирования ... 259
6.14.1. «Что» и «как» ... 260
6.14.2. Передача функций в качестве аргументов другим функциям, основы лямбда-выражений ... 261
6.14.3. Фильтрация, преобразование и свертка с библиотекой ranges (C++20) ... 263
6.15. Готовые объекты: шаблон класса vector стандартной библиотеки C++ ... 268
6.16. Итоги ... 275

Глава 7. Указатели в современном C++ (фактор риска) ... 277
7.1. Введение ... 278
7.2. Объявление и инициализация указателей ... 280
7.2.1. Объявление указателя ... 280
7.2.2. Инициализация указателя ... 280
7.2.3. Нулевые указатели до C++11 ... 281
7.3. Операторы для работы с указателями ... 281
7.3.1. Оператор взятия адреса (&) ... 281
7.3.2. Оператор косвенного обращения (*) ... 282
7.3.3. Использование операторов взятия адреса (&) и косвенного обращения (*) ... 283
7.4. Передача аргументов по указателю ... 284
7.5. Традиционные массивы в стиле C ... 286
7.5.1. Объявление традиционного массива и доступ к его элементам ... 286
7.5.2. Инициализация традиционного массива ... 289
7.5.3. Передача традиционного массива функции ... 289
7.5.4. Объявление параметров функции, обрабатывающей традиционный массив ... 289
7.5.5. Пример с функциями sort, begin и end стандартной библиотеки (C++11) ... 290
7.5.6. Недостатки традиционных массивов ... 290
7.6. Функция to_array (C++20), преобразующая традиционные массивы в std::array ... 291
7.7. Использование const с указателями и данными, на которые они указывают ... 293
7.7.1. Неконстантный указатель на неконстантные данные ... 293
7.7.2. Неконстантный указатель на константные данные ... 293
7.7.3. Константный указатель на неконстантные данные ... 294
7.7.4. Константный указатель на константные данные ... 295
7.8. Оператор sizeof ... 296
7.9. Выражения и арифметические операции с указателями ... 299
7.9.1. Прибавление и вычитание целых чисел ... 300
7.9.2. Вычитание одного указателя из другого ... 301

7.9.3. Присваивание указателя ... 301
7.9.4. Невозможность разыменования указателя void* ... 302
7.9.5. Сравнение указателей ... 302
7.10. Готовые объекты: доступ к контейнерам через span (C++20) ... 302
7.11. Строки на основе указателей ... 309
7.11.1. Аргументы командной строки ... 310
7.11.2. Еще одна возможность функции to_array (C++20) ... 312
7.12. Другие темы, связанные с указателями ... 313
7.13. Итоги ... 314

Глава 8. Строки и их представления, текстовые файлы, CSV-файлы и регулярные выражения ... 315
8.1. Введение ... 316
8.2. Присваивание и конкатенация строк ... 317
8.3. Сравнение строк ... 319
8.4. Создание подстроки ... 321
8.5. Обмен значениями двух строк ... 321
8.6. Получение характеристик строки ... 322
8.7. Поиск подстрок и символов в строке ... 325
8.8. Замена и удаление символов в строке ... 327
8.9. Вставка символов в строку ... 329
8.10. Преобразования между числами и строками (C++11) ... 330
8.11. Представления string_view (C++17) ... 332
8.12. Файлы и потоки ... 335
8.13. Создание файла с последовательным доступом ... 336
8.14. Чтение данных из файла с последовательным доступом ... 340
8.15. Чтение и запись текста в кавычках (C++14) ... 342
8.16. Обновление данных в файле с последовательным доступом ... 344
8.17. Обработка строковых потоков ... 344
8.18. Необработанные строковые литералы ... 347
8.19. Готовые объекты: чтение и анализ CSV-файла с данными о катастрофе «Титаника» ... 348
8.19.1. Чтение CSV-файла с помощью библиотеки rapidcsv ... 349
8.19.2. Работа с набором данных «Титаник» ... 351
8.20. Готовые объекты: основы регулярных выражений ... 359
8.20.1. Сравнение целых строк с шаблонами ... 361
8.20.2. Замена подстрок ... 365
8.20.3. Поиск в тексте по шаблонам ... 366
8.21. Итоги ... 369

Глава 9. Пользовательские классы ... 370
9.1. Введение ... 371
9.2. Тест-драйв объекта пользовательского класса Account ... 371
9.3. Класс Account с элементом данных, set-функцией и get-функцией ... 373
9.3.1. Определение класса ... 373
9.3.2. Спецификаторы доступа private и public ... 376
9.4. Конструктор класса Account ... 377
9.5. Разработка программ с set-функциями и get-функциями ... 381
9.6. Добавление «денежной» переменной в класс Account ... 383
9.7. Отделение интерфейса от реализации на примере класса Time ... 386
9.7.1. Интерфейс класса ... 388
9.7.2. Отделение интерфейса класса от реализации класса ... 388
9.7.3. Определение класса ... 389
9.7.4. Функции класса Time ... 390
9.7.5. Включение заголовка класса в файл исходного кода ... 391
9.7.6. Оператор разрешения области видимости (::) ... 391
9.7.7. Функция setTime и исключения ... 392
9.7.8. Функции to24HourString и to12HourString ... 392
9.7.9. Неявное встраивание функций ... 393
9.7.10. Функции класса в сравнении с глобальными функциями ... 393
9.7.11. Использование класса Time ... 393
9.7.12. Размер объекта ... 395
9.8. Процесс компиляции и компоновки программы ... 395
9.9. Область видимости класса и доступ к элементам класса ... 397
9.10. Функции доступа и служебные функции ... 398
9.11. Конструкторы с аргументами по умолчанию ... 398
9.11.1. Усовершенствованный класс Time ... 398
9.11.2. Перегрузка конструкторов и делегирование конструкторов (C++11) ... 404
9.12. Деструкторы ... 405
9.13. Вызов конструкторов и деструкторов ... 405
9.14. Коварная ловушка: возврат ссылки или указателя на скрытый элемент данных ... 409
9.15. Оператор присваивания по умолчанию ... 412
9.16. Константные объекты и константные функции класса ... 414
9.17. Композиция: объекты как элементы классов ... 416
9.18. Дружественные функции и классы ... 421
9.19. Указатель this ... 423
9.19.1. Неявное и явное использование указателя this для доступа к элементам данных объекта ... 424
9.19.2. Использование указателя this для каскадного вызова функций ... 425

9.20. Статические элементы класса: данные и функции, общие для всех объектов класса ... 429
9.21. Агрегаты ... 434
9.21.1. Инициализация агрегата ... 435
9.21.2. Назначенные инициализаторы (C++20) ... 435
9.22. Готовые объекты: сериализация в формате JSON ... 436
9.22.1. Сериализация вектора объектов, содержащего открытые данные ... 438
9.22.2. Сериализация вектора объектов, содержащего скрытые данные ... 442
9.23. Итоги ... 444

Глава 10. ООП: наследование и динамический полиморфизм ... 446
10.1. Введение ... 447
10.2. Базовые и производные классы ... 450
10.2.1. Иерархия класса CommunityMember ... 450
10.2.2. Иерархия класса Shape и открытое наследование ... 451
10.3. Взаимосвязь между базовым и производным классами ... 453
10.3.1. Объявление и использование класса SalariedEmployee ... 453
10.3.2. Создание иерархии наследования: SalariedEmployee и SalariedCommissionEmployee ... 456
10.4. Конструкторы и деструкторы в производных классах ... 462
10.5. Принцип динамического полиморфизма на примере объектов в видеоигре ... 463
10.6. Отношения между объектами в иерархии наследования ... 464
10.6.1. Вызов функций базового класса из объектов производного класса ... 465
10.6.2. Направление указателя производного класса на объект базового класса ... 468
10.6.3. Вызовы функций производного класса через указатель базового класса ... 469
10.7. Виртуальные функции и виртуальные деструкторы ... 470
10.7.1. Зачем нужны виртуальные функции ... 470
10.7.2. Объявление виртуальной функции ... 471
10.7.3. Вызов виртуальной функции ... 471
10.7.4. Виртуальные функции в иерархии SalariedEmployee ... 472
10.7.5. Виртуальные деструкторы ... 476
10.7.6. Финальные функции и классы ... 476
10.8. Абстрактные классы и чистые виртуальные функции ... 477
10.8.1. Чистые виртуальные функции ... 477
10.8.2. Драйверы устройств: полиморфизм в операционных системах ... 478
10.9. Полиморфизм в программе расчета зарплаты ... 479
10.9.1. Создание абстрактного базового класса Employee ... 480
10.9.2. Создание конкретного производного класса SalariedEmployee ... 482
10.9.3. Создание конкретного производного класса CommissionEmployee ... 485
10.9.4. Демонстрация полиморфных операций с объектами ... 487

10.10. Динамический полиморфизм, виртуальные функции и динамическое связывание «под капотом» компилятора ... 490
10.11. Идиома невиртуального интерфейса ... 493
10.12. Наследование интерфейса вместо наследования реализации ... 501
10.12.1. Переосмысление иерархии класса Employee: интерфейс CompensationModel ... 503
10.12.2. Класс Employee ... 503
10.12.3. Реализации CompensationModel ... 505
10.12.4. Тест-драйв новой иерархии ... 508
10.12.5. Преимущества внедрения зависимостей ... 509
10.13. Полиморфизм на основе шаблона класса std::variant и функции std::visit ... 509
10.14. Множественное наследование ... 516
10.14.1. Ромбовидное наследование ... 521
10.14.2. Устранение конфликта подобъектов с помощью виртуального наследования базового класса ... 523
10.15. Защищенные элементы класса ... 525
10.16. Открытое, защищенное и скрытое наследование ... 527
10.17. Другие методы динамического полиморфизма и статический полиморфизм ... 528
10.17.1. Другие методы динамического полиморфизма ... 529
10.17.2. Методы статического полиморфизма ... 530
10.17.3. Другие концепции полиморфизма ... 532
10.18. Итоги ... 533

Глава 11. Перегрузка операторов, семантика копирования и перемещения, умные указатели ... 535
11.1. Введение ... 536
11.2. Перегруженные операторы класса string стандартной библиотеки ... 538
11.3. Основные принципы перегрузки операторов ... 543
11.3.1. Перегрузка операторов не происходит автоматически ... 543
11.3.2. Операторы, которые не могут быть перегружены ... 544
11.3.3. Операторы, которые не нужно перегружать ... 544
11.3.4. Правила и ограничения перегрузки операторов ... 544
11.4. Управление динамической памятью с помощью операторов new и delete ... 545
11.5. Современные методы управления динамической памятью в C++: RAII и умные указатели ... 548
11.5.1. Умные указатели ... 549
11.5.2. Умный указатель unique_ptr ... 549
11.5.3. Владение объектами при работе с unique_ptr ... 551
11.5.4. Указатели unique_ptr и традиционные массивы ... 551

11.6. Создание класса myArray с перегрузкой операторов ... 552
11.6.1. Специальные функции класса ... 554
11.6.2. Работа с классом myArray ... 554
11.6.3. Определение класса myArray ... 565
11.6.4. Конструктор, задающий размер объекта myArray ... 566
11.6.5. Скобочная инициализация в конструкторе (C++11) ... 567
11.6.6. Копирующий конструктор и оператор копирующего присваивания ... 568
11.6.7. Перемещающий конструктор и оператор перемещающего присваивания ... 572
11.6.8. Деструктор ... 576
11.6.9. Функции toString и size ... 577
11.6.10. Перегрузка операторов равенства и неравенства ... 577
11.6.11. Перегрузка оператора индекса ... 579
11.6.12. Перегрузка унарного конвертирующего оператора bool ... 581
11.6.13. Перегрузка оператора префиксного инкремента ... 581
11.6.14. Перегрузка оператора постфиксного инкремента ... 582
11.6.15. Перегрузка оператора сложения с присваиванием ... 583
11.6.16. Перегрузка бинарных операторов извлечения из потока и вставки в поток ... 584
11.6.17. Дружественная функция swap ... 587
11.7. Оператор трехстороннего сравнения (C++20) ... 587
11.8. Преобразования между типами ... 591
11.9. Конструкторы и конвертирующие операторы со спецификатором explicit ... 592
11.10. Перегрузка оператора вызова функции ... 595
11.11. Итоги ... 595

Глава 12. Исключения и обзор контрактного программирования ... 597

12.1. Введение ... 598
12.2. Поток управления при обработке исключения ... 602
12.2.1. Определение класса исключений для представления типа проблемы, которая может возникнуть ... 602
12.2.2. Демонстрация обработки исключений ... 603
12.2.3. Включение кода в блок try ... 605
12.2.4. Определение обработчика catch для DivideByZeroException ... 605
12.2.5. Модель завершения обработки исключений ... 606
12.2.6. Поток управления при вводе ненулевого знаменателя ... 607
12.2.7. Поток управления при вводе нулевого знаменателя ... 607
12.3. Гарантии безопасности исключений и noexcept ... 608
12.4. Повторная генерация исключений ... 609
12.5. Размотка стека и необработанные исключения ... 611

12.6. Когда надо обрабатывать исключения ... 613
12.6.1. Макрос assert ... 616
12.6.2. Немедленное завершение ... 616
12.7. Конструкторы, деструкторы и обработка исключений ... 617
12.7.1. Генерация исключений в конструкторах ... 617
12.7.2. Перехват исключений в конструкторах с помощью функциональных блоков try ... 618
12.7.3. Исключения, деструкторы и noexcept(false) ... 620
12.8. Обработка исключений оператора new ... 621
12.8.1. Генерация исключения bad_alloc оператора new ... 622
12.8.2. Оператор new, возвращающий nullptr при сбое ... 623
12.8.3. Обработка исключений оператора new с помощью функции set_new_handler ... 624
12.9. Иерархия исключений стандартной библиотеки ... 625
12.10. Альтернатива блоку finally в языке C++: RAII («получение ресурсов есть инициализация») ... 628
12.11. Библиотеки, поддерживающие и исключения, и коды ошибок ... 629
12.12. Журнал ошибок ... 630
12.13. Основы контрактного программирования ... 631
12.14. Итоги ... 640

Глава 13. Контейнеры и итераторы стандартной библиотеки ... 642

13.1. Введение ... 643
13.2. Основы контейнеров ... 645
13.2.1. Общие вложенные типы в последовательных и ассоциативных контейнерах ... 647
13.2.2. Общие функции контейнеров ... 648
13.2.3. Требования к элементам контейнера ... 651
13.3. Работа с итераторами ... 652
13.3.1. Использование istream_iterator для ввода и ostream_iterator для вывода ... 652
13.3.2. Категории итераторов ... 654
13.3.3. Поддерживаемые контейнеры ... 655
13.3.4. Предопределенные типы итераторов ... 656
13.3.5. Операторы для работы с итераторами ... 656
13.4. Несколько слов об алгоритмах для обработки контейнеров ... 658
13.5. Последовательные контейнеры ... 658
13.6. Последовательный контейнер vector ... 659
13.6.1. Использование векторов и итераторов ... 659
13.6.2. Функции для обработки элементов вектора ... 664

13.7. Последовательный контейнер list ... 668
13.8. Последовательный контейнер deque ... 673
13.9. Ассоциативные контейнеры ... 675
13.9.1. Ассоциативный контейнер multiset ... 676
13.9.2. Ассоциативный контейнер set ... 680
13.9.3. Ассоциативный контейнер multimap ... 682
13.9.4. Ассоциативный контейнер map ... 685
13.10. Адаптеры контейнеров ... 686
13.10.1. Адаптер stack ... 687
13.10.2. Адаптер queue ... 689
13.10.3. Адаптер priority_queue ... 690
13.11. Псевдоконтейнер bitset ... 692
13.12. Основы нотации «О-большое» ... 694
13.13. Основы хеш-таблиц ... 697
13.14. Итоги ... 700

Глава 14. Алгоритмы стандартной библиотеки, диапазоны и представления C++20 ... 701

14.1. Введение ... 702
14.2. Требования алгоритмов: концепты (C++20) ... 704
14.3. Лямбда-выражения и алгоритмы ... 706
14.4. Алгоритмы ... 710
14.4.1. Алгоритмы fill, fill_n, generate и generate_n ... 710
14.4.2. Алгоритмы equal, mismatch и lexicographical_compare ... 713
14.4.3. Алгоритмы remove, remove_if, remove_copy и remove_copy_if ... 716
14.4.4. Алгоритмы replace, replace_if, replace_copy и replace_copy_if ... 721
14.4.5. Алгоритмы перемешивания, подсчетов, поиска минимального и максимального элементов ... 723
14.4.6. Алгоритмы поиска и сортировки ... 727
14.4.7. Алгоритмы swap, iter_swap и swap_ranges ... 731
14.4.8. Алгоритмы copy_backward, merge, unique, reverse, copy_if и copy_n ... 734
14.4.9. Алгоритмы inplace_merge, unique_copy и reverse_copy ... 737
14.4.10. Операции со множествами ... 739
14.4.11. Алгоритмы lower_bound, upper_bound и equal_range ... 742
14.4.12. Алгоритмы min, max и minmax ... 744
14.4.13. Алгоритмы gcd, lcm, iota, reduce и partial_sum (заголовок <numeric>) ... 745
14.4.14. Пирамидальная сортировка кучи и приоритетные очереди ... 749
14.5. Функциональные объекты (функторы) ... 754

14.6. Проекции ... 759
14.7. Представления (C++20) и функциональное программирование ... 763
14.7.1. Адаптеры диапазонов ... 763
14.7.2. Работа с адаптерами диапазонов и представлениями ... 765
14.8. Основы параллельных алгоритмов ... 770
14.9. Краткое описание алгоритмов стандартной библиотеки ... 772
14.10. Диапазоны в C++23 ... 776
14.11. Итоги ... 777

Глава 15. Шаблоны, концепты (C++20) и метапрограммирование ... 779
15.1. Введение ... 780
15.2. Пользовательские шаблоны классов и статический полиморфизм ... 783
15.3. Расширения функциональности шаблонов, введенные в C++20 ... 789
15.3.1. Сокращенные шаблоны функций (C++20) ... 789
15.3.2. Шаблоны лямбда-выражений (C++20) ... 791
15.4. Концепты (C++20): первый взгляд ... 791
15.4.1. Неограниченный шаблон функции multiply ... 793
15.4.2. Ограниченный шаблон функции с условием requires (C++20) ... 796
15.4.3. Концепты, предопределенные в C++20 ... 799
15.5. Признаки типов ... 801
15.6. Концепты: детальное рассмотрение ... 806
15.6.1. Создание пользовательского концепта ... 807
15.6.2. Использование концепта ... 808
15.6.3. Концепты в сокращенных шаблонах функций ... 809
15.6.4. Перегрузка на основе концептов ... 810
15.6.5. Выражения requires ... 813
15.6.6. Демонстрационные концепты ... 817
15.6.7. Технологии, применявшиеся до появления концептов: SFINAE и отправка тегов ... 818
15.7. Тест-драйв концептов с помощью объявления static_assert ... 819
15.8. Создание пользовательского алгоритма ... 822
15.9. Создание пользовательского контейнера и пользовательских итераторов ... 824
15.9.1. Шаблон класса ConstIterator ... 827
15.9.2. Шаблон класса Iterator ... 830
15.9.3. Шаблон класса MyArray ... 832
15.9.4. Дедуктивная инструкция для скобочной инициализации объекта myArray ... 836
15.9.5. Тест-драйв класса myArray и его пользовательских итераторов с алгоритмами std::ranges ... 838

15.10. Аргументы по умолчанию для параметров-типов шаблона ... 842
15.11. Шаблоны переменных ... 842
15.12. Вариативные шаблоны и выражения свертки ... 843
15.12.1. Кортежи: вариативный шаблон класса tuple ... 843
15.12.2. Вариативные шаблоны функций и основы выражений свертки (C++17) ... 847
15.12.3. Типы выражений свертки ... 851
15.12.4. Применение операторов в унарных выражениях свертки ... 852
15.12.5. Применение операторов в бинарных выражениях свертки ... 854
15.12.6. Оператор «запятая» для повторения операций ... 856
15.12.7. Ограничение элементов пакета параметров одним типом ... 857
15.13. Метапрограммирование на основе шаблонов ... 859
15.13.1. Шаблоны C++ обладают полнотой по Тьюрингу ... 860
15.13.2. Вычисления на этапе компиляции ... 861
15.13.3. Условная компиляция в метапрограммировании на основе шаблонов и инструкции constexpr if ... 866
15.13.4. Метафункции типа ... 869
15.14. Итоги ... 873

Глава 16. Модули [C++20]: технология разработки больших программ ... 875
16.1. Введение ... 876
16.2. Компиляция и компоновка до C++20 ... 878
16.3. Преимущества и предназначение модулей ... 880
16.4. Пример перехода на модули: заголовочные единицы трансляции ... 881
16.5. Модули могут уменьшать размеры единиц трансляции и время компиляции ... 884
16.6. Пример: создание и использование модуля ... 886
16.6.1. Объявление модульной единицы ... 887
16.6.2. Экспорт объявления ... 889
16.6.3. Экспорт группы объявлений ... 890
16.6.4. Экспорт пространства имен ... 890
16.6.5. Экспорт элемента пространства имен ... 891
16.6.6. Импорт модуля для его использования в экспорте объявлений ... 891
16.6.7. Пример: попытка доступа к неэкспортированному элементу модуля ... 893
16.7. Глобальный фрагмент модуля ... 896
16.8. Отделение интерфейса от реализации ... 896
16.8.1. Пример: единицы реализации модуля ... 897
16.8.2. Пример: класс на основе модулей ... 900
16.8.3. Фрагмент :private ... 903

16.9. Разделы модуля ... 904
16.9.1. Пример: разделы интерфейса модуля ... 904
16.9.2. Разделы реализации модуля ... 908
16.9.3. Пример: «подмодули» и разделы ... 908
16.10. Дополнительные примеры работы с модулями ... 913
16.10.1. Пример: импорт стандартной библиотеки C++ в виде модулей ... 914
16.10.2. Пример: запрет циклических зависимостей ... 916
16.10.3. Пример: инструкции import не наследуются ... 917
16.10.4. Пример: видимость и доступность модулей ... 918
16.11. Перенос кода в модули ... 920
16.12. Будущее модулей и инструментов для работы с ними ... 921
16.13. Итоги ... 922
Приложение: видео про модули ... 924
Приложение: статьи про модули ... 925
Приложение: глоссарий по модулям ... 928

Глава 17. Параллельные алгоритмы и конкурентность: высокоуровневый подход ... 931
17.1. Введение ... 932
17.2. Параллельные алгоритмы стандартной библиотеки (C++17) ... 935
17.2.1. Пример: сравнение быстродействия последовательного и параллельного алгоритмов сортировки ... 936
17.2.2. Когда нужны параллельные алгоритмы ... 940
17.2.3. Политики выполнения ... 941
17.2.4. Пример: сравнение быстродействия параллельных и векторных операций ... 941
17.2.5. Дополнительные сведения о параллельных алгоритмах ... 944
17.3. Многопоточное программирование ... 945
17.3.1. Состояния потока и жизненный цикл потока ... 945
17.3.2. Тупики и неопределенно долгие отсрочки ... 948
17.4. Запуск задач с помощью std::jthread ... 951
17.4.1. Определение задачи для выполнения в потоке ... 951
17.4.2. Выполнение задачи в потоке jthread ... 953
17.4.3. Преимущества класса jthread перед классом thread ... 956
17.5. Отношения между производителем и потребителем: первый эксперимент ... 957
17.6. Синхронизация доступа производителя и потребителя к общим изменяемым данным ... 965
17.6.1. Класс SynchronizedBuffer: мьютексы, блокировки и условные переменные ... 967
17.6.2. Тестирование класса SynchronizedBuffer ... 975

17.7. Производитель и потребитель: минимизация ожидания с помощью кольцевого буфера ... 979
17.8. Читатели и писатели ... 989
17.9. Кооперативное прерывание потоков класса jthread ... 991
17.10. Запуск задач с помощью шаблона функции std::async ... 994
17.11. Однократная потокобезопасная инициализация ... 1002
17.12. Основы атомарных типов ... 1004
17.13. Координация потоков с помощью защелок и барьеров (C++20) ... 1008
17.13.1. Защелки класса std::latch (C++20) ... 1009
17.13.2. Барьеры класса std::barrier (C++20) ... 1012
17.14. Семафоры (C++20) ... 1016
17.15. C++23: взгляд в будущее конкурентности ... 1020
17.15.1. Параллельные алгоритмы std::ranges ... 1020
17.15.2. Конкурентные контейнеры ... 1020
17.15.3. Другие материалы, связанные с конкурентностью ... 1021
17.16. Итоги ... 1021

Глава 18. Корутины (C++20) ... 1024
18.1. Введение ... 1025
18.2. Библиотеки поддержки корутин ... 1026
18.3. Установка библиотек concurrencpp и generator ... 1028
18.4. Создание корутины-генератора с помощью библиотеки generator ... 1029
18.5. Запуск задач с помощью библиотеки concurrencpp ... 1033
18.6. Создание корутины, содержащей co_await и co_return ... 1039
18.7. Низкоуровневые концепции корутин ... 1048
18.8. Развитие корутин в C++23 ... 1051
18.9. Итоги ... 1051

Приложение А. Приоритеты и группировка операторов ... 1052

Приложение Б. Набор символов ... 1055

Предисловие

Добро пожаловать в *«C++20 для программистов. Метод готовых объектов»*. В этой книге представлены передовые информационные технологии для разработчиков программного обеспечения. Она соответствует стандарту C++20 (1834 страницы), который был утвержден Комитетом ISO по стандартам C++ в сентябре 2020 года[1, 2].

Язык программирования C++ используется для создания высокопроизводительного, критически важного и надежного программного обеспечения. На нем пишут операционные системы, системы реального времени, встраиваемое ПО, видеоигры, банковское ПО, системы управления авиатранспортом, системы связи и многие другие программы. Эта книга представляет собой учебное пособие вводного и среднего уровня, предназначенное для освоения языка C++ (в том числе его стандартной библиотеки) в версии C++20, которая относится к самым популярным в мире языкам программирования. Мы представляем наглядный, современный, насыщенный примерами кода, ориентированный на практические задачи вводный курс C++20. Из этого предисловия вы узнаете о сути нашей книги.

Современный C++

В центре внимания — современный C++, включающий четыре последних стандарта: C++20, C++17, C++14 и C++11. Мы также упоминаем основные дополнения, которые могут появиться в C++23 или в более поздних версиях. Общей темой этой книги является акцент на новых, усовершенствованных способах написания кода на C++. Мы используем лучшие практики, делая акцент на современных идиомах современного профессионального программирования на C++, а также уделяем большое внимание вопросам производительности, безопасности и проектирования программного обеспечения.

---

[1] Финальный проект стандарта C++ опубликован по адресу https://timsong-cpp.github.io/cppwp/n4861. Эта версия бесплатна. Официальную версию в печатном виде (ISO/IEC 14882:2020) можно приобрести по адресу https://www.iso.org/standard/79358.html.

[2] Херб Саттер (Herb Sutter), «C++20 Approved, C++23 Meetings and Schedule Update», 6 сентября 2020 г. Был доступен: 15 сентября 2023 г. https://herbsutter.com/2020/09/06/c20-approved-c23-meetings-and-schedule-update.

В курсе событий

«Кто дерзает учить, должен никогда не переставать учиться»[1]. (Дж. К. Дана)

Чтобы «почувствовать пульс» современного языка C++, меняющего многие методы работы программистов, мы изучили около 6000 актуальных статей, исследовательских работ, технических описаний, документов, постов в блогах и форумах, а также видеопубликаций.

Кому пригодится эта книга

Книга *«C++20 для программистов. Метод готовых объектов»* рассчитана на следующие целевые аудитории:

- разработчики программ на C++, которые хотят изучить новейшие возможности C++20 по подробному учебнику, написанному в профессиональном стиле;
- разработчики программ на других языках, которые готовятся к выполнению проекта на языке C++ и хотят изучить его актуальную версию;
- разработчики, которые изучали C++ в вузе или работали с ним много лет назад, а теперь хотят освежить свои знания в контексте C++20;
- профессиональные преподаватели C++, составляющие курсы по C++20.

Готовые файлы с исходным кодом программ

В основе книги лежит фирменный подход Deitel — реальный код. Вместо отдельных фрагментов кода мы показываем язык C++ таким, каким он должен быть, на сотнях практических примеров готового кода, то есть работающих программ и выводимых ими данных.

Прочитайте раздел «Перед началом работы», который следует за этим предисловием, чтобы узнать, как подготовить свой компьютер с операционной системой Windows, macOS или Linux к выполнению более 200 приведенных в книге примеров кода, в которых около 15 тысяч строк. Для вашего удобства все эти примеры опубликованы на нашем веб-сайте и на портале GitHub в виде стандартных файлов C++ с расширениями `.cpp` и `.h`. Вы можете бесплатно загрузить их по любому из адресов[2]:

- https://github.com/pdeitel/cplusplus20forprogrammers
- https://www.deitel.com/books/c-plus-plus-20-for-programmers

---

[1] Джон Коттон Дана (John Cotton Dana). «В 1912 г. Дана, библиотекарь из Ньюарка, получил просьбу подобрать латинскую цитату для надписи на новом здании Государственного колледжа Ньюарка (ныне Университет Кин) в Юнионе (штат Нью-Джерси). Не найдя подходящей цитаты, Дана сочинил фразу, ставшую девизом колледжа. The New York Times Book Review, 5 марта 1967 г., с. 55. https://www.bartleby.com/73/1799.html.

[2] А также с сайта издательства «Питер», см. раздел «От издательства». — *Примеч. ред.*

- https://deitel.com/c-plus-plus-20-for-programmers
- https://deitel.com/cpp20fp

Все эти файлы готовы для компиляции как в интегрированных средах разработки, так и в компиляторах командной строки. Подробные инструкции о том, как компилировать и выполнять примеры кода с помощью трех рекомендуемых нами компиляторов, вы найдете в главе 1, в разделе 1.2. Компилируйте и запускайте соответствующие программы во время чтения книги, чтобы сделать учебный процесс более наглядным. Если у вас возникнут проблемы, вы можете связаться с нами по адресу:

deitel@deitel.com

Три профессиональных компилятора на выбор

Мы протестировали все приведенные в книге примеры кода в последних версиях трех компиляторов:

- Visual C++® в составе интегрированной среды разработки Microsoft® Visual Studio® Community Edition — в операционной системе Windows®;
- Clang C++ (clang++) в составе интегрированной среды разработки Apple® Xcode® — в операционной системе macOS®, а также в контейнере Docker®;
- GNU® C++ (g++) — в операционной системе Linux®, а также в контейнере Docker® с установленным пакетом GNU Compiler Collection (GCC).

Когда мы писали эту книгу, почти все языковые новшества, введенные в стандарт C++20, уже были полностью реализованы во всех трех компиляторах, некоторые — только в одном или двух и лишь немногие — ни в одном из трех. Мы указываем на эти различия при необходимости и будем обновлять наш цифровой контент по мере того, как эти новшества будут внедряться производителями компиляторов. Мы также будем публиковать обновления кода в репозитории этой книги на портале GitHub:

https://github.com/pdeitel/CPlusPlus20ForProgrammers

Кроме того, мы будем публиковать обновления кода и текста на веб-странице этой книги:

https://www.deitel.com/books/c-plus-plus-20-for-programmers

Учебный метод готовых объектов

В главе 9 мы расскажем о том, как разрабатывать на языке C++20 пользовательские (то есть ваши собственные) классы, а в последующих главах продолжим рассмотрение объектно-ориентированного программирования.

Что такое готовые объекты?

В первых главах вы будете работать с уже существующими классами, выполняющими достаточно сложные задачи. Используя всего несколько простых инструкций на языке C++, вы быстро создадите объекты этих классов и заставите их «показать свою силу». Мы называем это методом готовых объектов.

Учитывая огромное количество бесплатных библиотек классов с открытым исходным кодом (open-source), созданных сообществом разработчиков C++, вы сможете выполнять сложные задачи задолго до того, как научитесь создавать свои собственные классы в главе 9. Это один из наиболее привлекательных аспектов работы с объектно-ориентированными языками, особенно с таким совершенным объектно-ориентированным языком, как C++.

Бесплатные классы

Мы обращаем ваше внимание, что в «экосистеме» C++ есть огромное количество очень полезных и совершенно бесплатных классов, доступных программистам. Основными источниками этих классов являются:

- стандартная библиотека C++;
- библиотеки, специфичные для конкретной платформы, например поставляемые с Microsoft Windows, Apple macOS или различными версиями Linux;
- бесплатные сторонние библиотеки C++, создаваемые сообществом разработчиков открытого программного обеспечения;
- коллеги-программисты, например сотрудники вашей организации.

Мы рекомендуем вам черпать вдохновение для ваших собственных программ, изучая многочисленные общедоступные программы на языке C++, публикуемые в виде открытого исходного кода на таких сайтах, как GitHub.

Проект Boost

Проект Boost предоставляет программистам 168 библиотек C++ с открытым исходным кодом[1]. Кроме того, в нем «выращиваются» многие языковые новшества, которые впоследствии включаются в стандартную библиотеку C++. Например, именно из проекта Boost берут начало уже вошедшие в современный C++ средства многопоточности и генерации случайных чисел, интеллектуальные указатели, кортежи, регулярные выражения, файловые системы и класс `string_view`[2]. На этой странице сайта Stack Overflow приведен список биб-

[1] «Boost 1.78.0 Library Documentation». Был доступен: 15 сентября 2023 г. https://www.boost.org/doc/libs/1_78_0.

[2] «Boost (C++ libraries)». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Boost_(C++_libraries).

лиотек и языковых средств современного C++, которые эволюционировали из библиотек Boost[1]:

https://stackoverflow.com/a/8852421

Примеры кода с готовыми объектами

В главе 1 кратко рассматриваются основные понятия и термины объектной технологии. В главах 2–8 вы будете создавать и использовать объекты уже существующих классов (задолго до создания своих собственных классов в главе 9 и далее). Наши примеры кода с готовыми объектами посвящены следующим учебным темам:

- Раздел 2.7 — создание и использование объектов класса `string` стандартной библиотеки.
- Раздел 3.12 — работа с очень большими целыми числами.
- Раздел 4.13 — создание и чтение ZIP-файлов.
- Раздел 5.20 — *Lnfylun Lhqtomh Wjtz Qarcv: Qjwazkrplm xzz Xndmwwqhlz* (это зашифрованная фраза, речь идет о шифровании с закрытым ключом).
- Раздел 6.15 — шаблон класса `vector` стандартной библиотеки C++.
- Раздел 7.10 — доступ к контейнерам с данными с помощью класса `span` (C++20).
- Раздел 8.19 — чтение и анализ CSV-файла со статистическими данными о катастрофе «Титаника».
- Раздел 8.20 — основы регулярных выражений.
- Раздел 9.22 — сериализация данных в формате JSON (JavaScript Object Notation).

Прекрасный пример метода готовых объектов — использование объектов таких стандартных классов C++, как `array` и `vector` (глава 6). Вы можете работать с ними, даже не зная, как они устроены, и вообще не умея писать пользовательские классы. На протяжении всей книги мы используем обширные возможности стандартной библиотеки C++.

Краткое содержание книги

На внутренней стороне обложки книги показана в виде наглядной схемы ее модульная структура. Читая раздел «Краткое содержание книги», сверяйтесь с этой схемой, которая поможет быстрее ознакомиться с темами глав и разделов.

---

[1] Kennytm, ответ на вопрос «Which Boost Features Overlap with C++11?». Был доступен: 15 сентября 2023 г. https://stackoverflow.com/a/8852421.

В первых главах книги закладывается прочный фундамент основ языка C++20. Последующие главы с более сложными практическими примерами кода помогут вам перейти к разработке ваших собственных программ на C++20. На протяжении всей книги рассматриваются пять моделей программирования на C++20:

- процедурное программирование;
- функциональное программирование;
- объектно-ориентированное программирование;
- обобщенное программирование;
- метапрограммирование на основе шаблонов.

Часть 1. Быстрый старт и базовые возможности C++

Глава 1 «Введение и тест-драйв популярных бесплатных компиляторов C++» рассказывает о том, что должен знать каждый профессиональный программист. В этой главе вы найдете:

- краткое введение;
- обсуждение закона Мура, многоядерных процессоров и необходимости стандартизированного конкурентного программирования в современном C++;
- краткую справку по объектно-ориентированному программированию с объяснением терминов, которые встретятся вам в последующих главах.

В этой главе вы также узнаете, как компилировать и выполнять написанный на C++ готовый код с помощью трех бесплатных компиляторов, которые мы вам рекомендуем:

- Visual C++ из Microsoft Visual Studio (в Windows);
- Apple Xcode (в macOS);
- GNU g++ (в Linux).

Мы проверили на каждом из них все приведенные в книге примеры кода, прокомментировав те немногие случаи, когда наши версии компиляторов (существующие на момент выхода книги в 2022 году) еще не поддерживают какие-либо из новшеств стандарта C++20. Выберите ту среду разработки программ, которая вам больше нравится. Наши примеры кода будут хорошо работать и с другими компиляторами C++20.

Мы расскажем, как запускать компиляторы командной строки GNU g++ и clang++ в контейнерах Docker, чтобы вы могли работать с самыми современными версиями этих компиляторов в любой из трех операционных систем: Windows, macOS или Linux. Docker — это ценный инструмент разработчика, дополнительные сведения о нем приведены в разделе «Перед началом работы». Кроме того, мы расскажем, как установить Linux на компьютер с Windows, используя подсистему Windows для Linux (WSL). Это еще один способ работы в Windows с компиляторами g++ и clang++.

В главе 2 «Азы программирования на C++» представлены базовые понятия языка C++, такие как ввод и вывод данных, основные типы данных, арифметические операторы и их приоритеты, инструкции принятия решений. В разделе 2.7 вы познакомитесь с методом готовых объектов, легко создавая и используя объекты класса `string` стандартной библиотеки (для этого вам не надо знать, как устроен этот большой и сложный класс и как вообще устроены классы).

Глава 3 «Управляющие инструкции (часть 1)» посвящена некоторым управляющим инструкциям. В ней вы изучите инструкции выбора `if` и `if...else`, инструкции цикла `while` со счетчиком и контрольным значением, а также операторы инкремента, декремента и присваивания. В разделе 3.12 мы покажем очередной пример использования готовых объектов: с помощью общедоступного класса вы будете выполнять вычисления с чрезвычайно большими целыми числами.

Глава 4 «Управляющие инструкции (часть 2)» рассказывает об управляющих инструкциях `for`, `do...while`, `switch`, `break` и `continue`, а также о логических операторах. В разделе 4.13 вы продолжите работать с готовыми объектами, включив библиотеку `miniz-cpp` для записи и чтения ZIP-файлов.

В главе 5 «Функции и шаблоны функций» представлены пользовательские функции. Мы расскажем о методах моделирования на примере генерации случайных чисел. Устаревшая функция генерации случайных чисел `rand`, которую C++ унаследовал от языка C, может быть предсказуемой, поэтому программы с ней ненадежны. Мы рассмотрим более безопасную библиотеку, введенную в C++11, которая генерирует недетерминированные (непредсказуемые) последовательности случайных чисел. Такие генераторы случайных чисел используются для моделирования событий и в сценариях безопасности, где предсказуемость недопустима. Также мы обсудим передачу информации между функциями и рекурсию. Пример с готовыми объектами в разделе 5.20 продемонстрирует шифрование с закрытым ключом.

Часть 2. Массивы, указатели и строки

В главе 6 «Массивы, векторы, библиотека ranges и функциональное программирование» мы приступим к рассмотрению контейнеров, итераторов и алгоритмов стандартной библиотеки C++. Мы представим вам контейнер `array`, предназначенный для хранения и обработки списков и таблиц значений. Вы определите массивы класса `array`, инициализируете их и получите доступ к их элементам. Вы узнаете, как передавать эти массивы функциям, выполнять сортировку и поиск в массивах, работать с многомерными массивами. Мы познакомим вас с функциональным программированием, начав с лямбда-выражений (безымянных функций) и диапазонов библиотеки `ranges`, которая входит в «большую четверку» новых возможностей C++20. В разделе 6.15 на примере готовых объектов мы продемонстрируем векторы — контейнеры данных, основанные на шаблоне класса `vector` стандартной библиотеки C++. По сути, вся эта глава представляет собой большой урок работы с готовыми объектами — массивами и векторами. Примеры исходного кода в этой главе хорошо иллюстрируют современные идиомы программирования на C++.

В главе 7 «Указатели в современном C++ (фактор риска)» подробно рассматриваются указатели и объясняется тесная взаимосвязь между указателями, массивами на основе указателей и строками на основе указателей (которые также называют строками в стиле C). Все эти традиционные элементы C++ унаследовал от языка C. Указатели являются мощным инструментом программиста, но с ними сложно работать и они часто становятся причиной ошибок в программах. Поэтому мы покажем вам современные альтернативы указателям, которые сделают ваш код более надежным и безопасным: классы `array` и `vector`, класс `span` (C++20) и класс `string_view` (C++17). Но вы должны уметь работать и с традиционными указателями, чтобы читать код старых программ. Мы расскажем о тех немногих случаях, когда указатели нужны даже в новых программах. Во всех остальных случаях мы рекомендуем использовать вместо указателей современные языковые средства. К ним относится класс `span`, с объектами которого вы будете работать в разделе 7.10. С помощью этого класса вы сможете, не используя указатели, читать и обрабатывать элементы традиционных массивов на основе указателей, а также современных контейнеров `array` и `vector`. В этой главе мы продолжим акцентировать внимание на современных идиомах программирования на C++.

В главе 8 «Строки и их представления, текстовые файлы, CSV-файлы и регулярные выражения» вы узнаете о больших возможностях класса `string` стандартной библиотеки. Затем мы расскажем, как выполнять операции чтения и записи с обычными текстовыми файлами и с файлами в формате CSV, которые часто применяются для работы с наборами данных. Мы также расскажем об операциях со строками с помощью поддерживаемых стандартной библиотекой регулярных выражений. В C++ есть два типа строк: объекты класса `string` и традиционные строки на основе указателей. Мы используем объекты класса `string`, чтобы сделать программы более надежными и избежать многих проблем безопасности, свойственных традиционным строкам. В новых разработках надо отдавать предпочтение объектам класса `string`. Вы узнаете о классе `string_view` (C++17) — простом и универсальном средстве передачи строк любого типа функциям. В конце главы вы продолжите работать с готовыми объектами:

- Раздел 8.19 посвящен анализу данных. Вы прочитаете и проанализируете CSV-файл, содержащий набор данных «Титаник», — один из популярных учебных инструментов для начинающих аналитиков.
- Раздел 8.20 посвящен регулярным выражениям, которые нужны для поиска и замены фрагментов текста.

Часть 3. Объектно-ориентированное программирование

В главе 9 «Пользовательские классы» вы начнете изучать объектно-ориентированное программирование и узнаете, как создавать свои собственные классы. Язык C++ является расширяемым: каждый созданный вами класс становится новым типом данных, после чего вы можете создавать любые объекты этого типа. В разделе 9.22 вы найдете очередной пример работы с готовыми объектами: сторонняя библиотека `cereal` позволит вам легко выполнять сериализацию (преобра-

зование объектов в формат JavaScript Object Notation — JSON) и десериализацию (обратное преобразование из формата JSON).

Глава 10 «ООП: наследование и динамический полиморфизм» посвящена отношениям между классами в иерархии наследования и мощным средствам динамического полиморфизма, обеспечивающим эти отношения. Из этой главы вы узнаете об основных принципах полиморфизма. Один из самых важных моментов этой главы — подробная схема, объясняющая, как полиморфизм, виртуальные функции и динамическое связывание реализованы «под капотом» языка C++. Вы увидите, что для этого используется элегантная структура данных на основе указателей. Мы расскажем и о других средствах полиморфизма, среди которых идиома невиртуального интерфейса (NVI), а также `std::variant` и `std::visit`. Вы узнаете о преимуществах наследования интерфейса перед наследованием реализации.

В главе 11 «Перегрузка операторов, семантика копирования и перемещения, умные указатели» вы узнаете, как использовать стандартные операторы языка C++ для операций с объектами пользовательских (ваших собственных) классов. Вы также познакомитесь с умными указателями и с динамическим управлением памятью. Умные указатели позволяют избежать ошибок динамического управления памятью, предоставляя программисту больше возможностей, чем традиционные указатели. Мы обсудим умные указатели класса `unique_ptr`. Ключевая тема этой главы — создание полноценных высококачественных классов. Мы начнем с тест-драйва класса `string`, который является красивым примером перегрузки операторов. Затем мы перейдем к одному из важнейших примеров книги: вы узнаете, как написать пользовательский класс контейнеров `MyArray`, используя перегруженные операторы и другие языковые средства таким образом, чтобы в вашем классе не было проблем, свойственных массивам на основе указателей[1]. Мы представим вам и реализуем пять специальных функций, которые можно определять в каждом классе: копирующий конструктор, оператор копирующего присваивания, перемещающий конструктор, оператор перемещающего присваивания и деструктор. Мы обсудим семантику копирования, а также семантику перемещения, которая позволяет компилятору быстро перемещать ресурсы из одного объекта в другой, избегая бесполезных и ресурсоемких операций копирования. Мы расскажем о новом операторе трехстороннего сравнения (`<=>`) и о том, как реализовать пользовательские конвертирующие операторы. В главе 15 вы усовершенствуете класс `MyArray`, преобразовав его в шаблон класса, позволяющий хранить данные разных типов. Вы создадите по-настоящему полезные классы.

Глава 12 «Исключения и обзор контрактного программирования» продолжает тему обработки исключений, начатую в главе 6. Мы объясним, когда нужны исключения и какие бывают гарантии безопасности исключений, рассмотрим

[1] В программах для промышленного применения вы должны использовать для контейнеров классы стандартной библиотеки, но этот учебный пример позволяет продемонстрировать многие ключевые концепции современного C++.

исключения в контексте конструкторов и деструкторов, расскажем об обработке ошибок динамического выделения памяти и о том, почему в некоторых проектах исключения запрещены. Глава завершается введением в так называемые контракты, которые могут появиться в будущих версиях C++; мы расскажем о контрактах на примере их экспериментальной реализации, опубликованной на сайте godbolt.org. Благодаря контрактам программисты смогут отказаться от большинства исключений (то есть объявлять большинство функций со спецификатором `noexcept`), что позволит компилятору оптимизировать исполняемый код и избежать сложных ресурсоемких операций, связанных с обработкой исключений.

Часть 4. Контейнеры, итераторы и алгоритмы стандартной библиотеки

В главе 13 «Контейнеры и итераторы стандартной библиотеки» начинается более широкое и глубокое рассмотрение трех ключевых компонентов стандартной библиотеки языка C++:

- контейнеров (основанных на шаблонах структур данных);
- итераторов (необходимых для доступа к элементам контейнеров);
- алгоритмов (использующих итераторы для обработки контейнеров).

Мы обсудим контейнеры, а также адаптеры контейнеров и псевдоконтейнеры. Вы увидите, что стандартная библиотека C++ предоставляет вам готовые структуры данных, широко применяемые программистами, поэтому вам не нужно создавать свои собственные. Почти все связанные со структурами данных задачи, которые вам могут встретиться, вы сможете решить средствами стандартной библиотеки. Мы продемонстрируем большинство контейнеров стандартной библиотеки и объясним, как применять алгоритмы к различным типам контейнеров, используя итераторы. Вы увидите, что разные контейнеры поддерживают разные типы итераторов. Мы продолжим показывать те ценные возможности упрощения кода, которые дает программисту введенная в C++20 библиотека `ranges`.

В главе 14 «Алгоритмы стандартной библиотеки, диапазоны и представления C++20» представлены многие из 115 алгоритмов стандартной библиотеки. В центре внимания — типичные операции с контейнерами, в том числе заполнение контейнеров значениями, генерация значений, сравнение элементов и целых контейнеров, удаление элементов, замена элементов, математические операции, поиск, сортировка, обмен значениями, копирование, объединение, операции со множествами, определение границ контейнеров, нахождение наименьших и наибольших значений. Мы обсудим минимальные требования к итераторам, чтобы вы знали, какие контейнеры совместимы с тем или иным алгоритмом. Мы начинаем обсуждение концептов, входящих в «большую четверку» новых возможностей C++20. Алгоритмы из пространства имен `std::ranges` (C++20) используют концепты, чтобы проверять корректность получаемых аргументов. Мы продолжим обсуждение функционального программирования на примере диапазонов и представлений C++20.

Часть 5. Более сложные темы

В главе 15 «Шаблоны, концепты (C++20) и метапрограммирование» рассматривается обобщенное программирование на основе шаблонов, которое применяется в C++ с момента выхода стандарта 1998 года. Роль шаблонов возрастала с каждым новым выпуском C++. Для современного C++ характерен перенос значительной части ресурсоемких операций на этап компиляции, что позволяет компилятору более надежно контролировать типы данных и повышает быстродействие во время выполнения программ. Как вы увидите, шаблоны и особенно метапрограммирование на основе шаблонов — это важнейшие средства для переноса многих операций с этапа выполнения на этап компиляции. В этой главе мы более подробно рассмотрим шаблоны, объясним, как разрабатывать ваши собственные шаблоны классов, и детально рассмотрим введенные в C++20 концепты. Вы создадите свои собственные концепты, преобразуете созданный в главе 11 класс `MyArray` в шаблон класса, имеющий итераторы, а также поработаете с вариативными шаблонами, которые могут принимать любое количество аргументов. Мы познакомим вас с метапрограммированием на основе шаблонов.

Глава 16 «Модули C++20: технология разработки больших программ» представляет следующую из «большой четверки» новых возможностей C++20. Модули — это новый способ структурирования кода, позволяющий вам решать, какие объявления надо открыть для клиентского кода, и предотвратить доступ извне ко всем остальным деталям реализации классов. Модули помогают программистам работать эффективнее, особенно при создании, сопровождении и модернизации больших информационных систем. Модули помогают быстрее создавать такие системы и делают их более масштабируемыми. По словам создателя C++ Бьерна Страуструпа, «модули дают языку C++ историческую возможность улучшить чистоту кода и время компиляции (перенося C++ в XXI век)». Вы увидите, что даже при разработке небольших систем модули немедленно дают преимущества, устраняя необходимость в препроцессоре C++. Мы бы с удовольствием интегрировали модули в наши программы, но на данный момент в наших основных компиляторах еще не реализована полноценная поддержка модулей.

Глава 17 «Параллельные алгоритмы и конкурентность: высокоуровневый подход» — одна из самых важных в книге. В ней представлены имеющиеся в C++ средства создания многопоточных приложений. Распределение нескольких задач между потоками процессора может значительно повышать быстродействие и скорость реакции программы. Мы показываем, как использовать готовые параллельные алгоритмы (C++17) для создания многопоточных программ, которые будут работать быстрее (как правило, намного быстрее) на современных компьютерах с многоядерными процессорами. Например, мы сортируем 100 миллионов чисел сначала последовательным, а затем параллельным алгоритмом, измеряя с помощью библиотеки `<chrono>` прирост производительности параллельного алгоритма на 4-ядерном и 8-ядерном процессорах. Вы увидите, что на компьютере с 64-битной версией Windows 10 и 8-ядерным процессором Intel параллельная сортировка может выполняться почти в 7 раз быстрее, чем последовательная. Мы обсуждаем взаимодействие потока-производителя и пото-

ка-потребителя, демонстрируя различные способы их синхронизации с помощью низкоуровневых и высокоуровневых средств конкурентного программирования, в том числе защелок, барьеров и семафоров, введенных в C++20. Мы объясняем, в чем заключается сложность конкурентного программирования и почему при любой возможности надо отдавать предпочтение высокоуровневым средствам конкурентности. Такие низкоуровневые средства, как семафоры и атомарные типы, имеет смысл применять для реализации средств более высокого уровня, таких как защелки.

В главе 18 «Корутины (C++20)» представлены корутины — последняя из «большой четверки» новых возможностей C++20. Корутина — это функция, которая может приостановить свое выполнение, а впоследствии возобновить его по инструкции из другой части программы. Все механизмы реализации такого поведения корутин полностью реализуются компилятором без вашего участия. Вы увидите, что корутиной является любая функция, содержащая хотя бы одно из ключевых слов `co_await`, `co_yield` или `co_return`, и такие функции дают вам возможность писать программы с конкурентными задачами, сохраняя простой последовательный стиль кодирования. Для поддержки корутин нужна сложная инфраструктура, которую вы теоретически могли бы написать самостоятельно, но это сложно, утомительно и чревато ошибками. Большинство экспертов сходятся во мнении, что надо использовать готовые высокоуровневые библиотеки поддержки корутин, и мы демонстрируем именно такой подход. В сообществе разработчиков открытого программного обеспечения уже созданы несколько экспериментальных библиотек для быстрой и удобной работы с корутинами. Две из них мы используем в примерах кода этой главы. Вероятно, поддержка корутин будет добавлена в стандартную библиотеку C++23.

Приложения

Приложение А «Приоритеты и группировка операторов» содержит список стандартных операторов языка C++, отсортированных по их приоритетам. В первый раздел списка включены наиболее приоритетные операторы (выполняемые первыми), в каждый последующий раздел — операторы с последовательно убывающими приоритетами.

Приложение Б «Набор символов ASCII» содержит список стандартных символов и соответствующих им чисел.

Дополнительные материалы на сайте deitel.com

Веб-страница этой книги расположена на сайте deitel.com по адресу:

https://deitel.com/c-plus-plus-20-for-programmers

На этой странице вы найдете следующие дополнительные материалы:

- ссылки на наш репозиторий на портале GitHub, содержащий все примеры исходного кода из этой книги в виде загружаемых файлов;

- посты в блоге: https://deitel.com/blog;
- обновления к книге.

Рекомендации по загрузке файлов и по настройке вашей среды разработки программ на C++ приведены в разделе «Перед началом работы».

C++ Core Guidelines

C++ Core Guidelines — это сборник рекомендаций объемом около 500 печатных страниц, «призванный помочь людям эффективно использовать современный язык C++»[1]. Он написан под редакцией Бьерна Страуструпа (создателя языка C++) и Херба Саттера (председателя комитета ISO по стандартам C++):

https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

Авторы C++ Core Guidelines указывают:

> *Эти рекомендации сосредоточены на относительно высокоуровневых проблемах, таких как интерфейсы, управление ресурсами, управление памятью и конкурентность. Эти правила влияют на архитектуру приложений и на проектирование библиотек. Результатом выполнения этих правил будет статически типобезопасный код, не вызывающий утечек ресурсов, в котором будет исправлено значительно больше логических ошибок программирования, чем обычно бывает в современном коде. И он будет работать быстро — вы сможете позволить себе делать все правильно*[2].

На протяжении всей книги мы используем соответствующие рекомендации из C++ Core Guidelines. Вы обязательно должны уделять пристальное внимание этим мудрым советам. В сборник включены сотни основных рекомендаций, разделенных на десятки категорий и подкатегорий. Их выполнение может показаться вам крайне трудным, но с помощью статических анализаторов кода (см. далее) вы легко проверите любой свой код на соответствие C++ Core Guidelines.

Библиотека Guidelines Support Library (GSL)

В C++ Core Guidelines часто упоминается библиотека Guidelines Support Library (GSL), в которой реализованы вспомогательные классы и функции для поддержки различных рекомендаций[3]. Microsoft предоставляет свою реализацию GSL с открытым исходным кодом на портале GitHub по адресу:

https://github.com/Microsoft/GSL

---

[1] C++ Core Guidelines, «Abstract». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-abstract.

[2] C++ Core Guidelines, «Abstract».

[3] C++ Core Guidelines, «GSL: Guidelines Support Library». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-gsl.

Мы используем GSL в нескольких примерах кода из первой половины книги. Некоторые средства из GSL впоследствии были включены в стандартную библиотеку C++.

Профессиональные статические анализаторы кода

Эти средства статического анализа кода позволяют быстро проверить ваш код на наличие типичных ошибок и проблем безопасности, а также сообщают о том, как его улучшить. Использование анализаторов дает почти такой же эффект, как проверка кода экспертами мирового класса. Чтобы всегда следовать рекомендациям C++ Core Guidelines и улучшить приведенные в книге примеры кода, мы использовали следующие статические анализаторы:

- `clang-tidy` — https://clang.llvm.org/extra/clang-tidy;
- `cppcheck` — https://cppcheck.sourceforge.io;
- средства проверки кода на соответствие C++ Core Guidelines, встроенные в статический анализатор кода Microsoft Visual Studio.

С помощью этих трех инструментов мы проверили:

- соответствие рекомендациям C++ Core Guidelines;
- соответствие стандартам кодирования;
- соответствие современным идиомам C++;
- отсутствие проблем с безопасностью;
- отсутствие типичных ошибок;
- отсутствие проблем с быстродействием;
- наглядность кода

и многое другое.

Кроме того, мы использовали флаг `-Wall` в командах компиляторов GNU g++ и Clang C++, чтобы включить все категории выдаваемых ими предупреждений, сообщающих о возможных проблемах. Мы добились того, что наши программы компилируются без таких сообщений (кроме редких случаев, выходящих за рамки тем этой книги). Информация о настройке средств статического анализа кода приведена в разделе «Перед началом работы».

Интернет-ресурсы для программистов

Stack Overflow

Stack Overflow — один из популярнейших сайтов вопросов и ответов, предназначенный для разработчиков программного обеспечения. Многие проблемы,

с которыми сталкиваются программисты, уже обсуждались на Stack Overflow, поэтому это отличное место, чтобы искать решения таких проблем и задавать новые вопросы. В ходе работы над книгой многие из наших поисковых запросов в Google на различные темы, зачастую сложные, приводили среди первых результатов ответы на Stack Overflow.

GitHub

> *Лучший способ подготовиться [к работе программиста] — это писать программы и изучать замечательные программы, написанные другими людьми. В моем случае я ходил к мусорным корзинам в Центре компьютерных наук и вылавливал из них листинги операционных систем*[1]. (Б. Гейтс)

GitHub — это отличная площадка для поиска бесплатного и открытого исходного кода, который может пригодиться вам в ваших проектах. Если хотите, вы можете публиковать на GitHub исходный код написанных вами программ, участвуя в деятельности всемирного сообщества разработчиков открытого программного обеспечения. Пятьдесят миллионов программистов используют GitHub[2]. На сайте размещено более 200 миллионов репозиториев исходного кода на огромном количестве языков программирования[3]. В 2021 году авторы опубликовали свои новые работы более чем в 61 миллионе репозиториев[4]. GitHub является важнейшим элементом арсенала профессионального разработчика программного обеспечения и содержит средства контроля версий, которые помогают командам разработчиков управлять проектами с открытым кодом и частными проектами.

Сообщество разработчиков C++ с открытым кодом очень велико. На GitHub есть более 41 тысячи[5] репозиториев исходного кода на C++. На GitHub вы можете исследовать написанный другими код и при желании компилировать его. Это отличный способ изучения программирования, который является естественным продолжением нашего учебного подхода на основе реального кода[6].

В 2018 году компания Microsoft приобрела GitHub за 7.5 миллиарда долларов. Если вы профессиональный программист, вы почти наверняка регулярно пользуетесь GitHub. По словам генерального директора Microsoft Сатьи Наделлы, компания приобрела GitHub, чтобы «расширить для каждого разработчика

---

[1] Билл Гейтс (Bill Gates), цитата из работы Сьюзан Ламмерс (Susan Lammers) «Interviews with 19 Programmers Who Shaped the Computer Industry». Microsoft Press, 1986, с. 83.

[2] «GitHub». Был доступен: 15 сентября 2023 г. https://github.com.

[3] «Where the World Builds Software». Был доступен: 15 сентября 2023 г. https://github.com/about.

[4] «The 2021 State of the Octoverse». Был доступен: 15 сентября 2023 г. https://octoverse.github.com.

[5] «C++». Был доступен: 15 сентября 2023 г. https://github.com/topics/cpp.

[6] Перед использованием публикуемых на GitHub программ с открытым исходным кодом вам необходимо ознакомиться с разнообразными лицензиями на такие программы.

возможности творчества, внедрения инноваций и решения самых актуальных мировых проблем»[1].

Мы призываем вас изучать и компилировать многочисленные программы с портала GitHub, а также вносить свой вклад в его работу.

Docker

Мы используем Docker — инструмент для упаковки программ в контейнеры, содержащие все необходимое для того, чтобы было удобно работать с этими программами на компьютерах с разными операционными системами. Некоторые программы и программные пакеты трудно устанавливать и настраивать. Но многие из них можно бесплатно загрузить в виде готовых контейнеров Docker, чтобы избежать сложных проблем с установкой. Сразу после загрузки вы можете запускать их на обычных настольных компьютерах или ноутбуках, что делает Docker отличным средством для быстрого, удобного и экономичного изучения новых технологий.

Мы покажем, как устанавливать и запускать полностью настроенные контейнеры Docker, содержащие:

- пакет компиляторов GNU Compiler Collection (GCC), в состав которого входит компилятор g++;
- последнюю версию компилятора clang++ от команды разработчиков Clang.

Эти контейнеры работают в Docker на компьютерах с Windows, macOS и Linux.

Docker способствует воспроизводимости результатов. В контейнерах Docker вы можете сохранять ваши настроенные конфигурации со всеми программами и библиотеками, которые используете. Это позволит другим программистам полностью воссоздавать вашу рабочую среду, а затем точно повторять вашу работу. И вы тоже сможете таким образом стабильно воспроизводить ваши собственные результаты. Воспроизводимость особенно важна в науке и в медицине, например, когда исследователи хотят подтвердить и расширить результаты работ, изложенные в опубликованных статьях.

Некоторые ключевые документы и ресурсы по C++

В книгу включены сотни ссылок на видеоролики, записи в блогах, статьи и онлайн-справочники, которые мы изучали, работая над книгой. Возможно, вы захотите обратиться к некоторым из этих ресурсов для изучения более сложных языковых средств и идиом. Основным сборником документации по C++ фактически является сайт cppreference.com. Мы часто ссылаемся на него, чтобы у вас была подробная справочная информация о стандартных классах и функциях C++, рассматрива-

[1] «Microsoft to Acquire GitHub for $7.5 Billion». Был доступен: 15 сентября 2023 г. https://news.microsoft.

емых в книге. Кроме того, мы часто ссылаемся на финальный проект стандарта C++20, который доступен бесплатно на GitHub по адресу:

https://timsong-cpp.github.io/cppwp/n4861

Ниже мы приводим список ресурсов по C++, которые могут быть вам полезны в процессе изучения нашей книги.

Документация

- Финальный проект стандарта C++20, принятый Комитетом по стандартам C++:

  https://timsong-cpp.github.io/cppwp/n4861

- Документация по C++:

  https://cppreference.com

- Документация по C++ от Microsoft:

  https://docs.microsoft.com/en-us/cpp/cpp

- Документация по стандартной библиотеке GNU C++:

  https://gcc.gnu.org/onlinedocs/libstdc++/manual/index.html

Блоги

- Блог Херба Саттера (Herb Sutter) «Sutter's Mill» о разработке программного обеспечения:

  https://herbsutter.com

- Блог команды Microsoft по C++:

  https://devblogs.microsoft.com/cppblog

- Блог Мариуса Банчилы (Marius Bancila):

  https://mariusbancila.ro/blog

- Блог Джонатана Боккары (Jonathan Boccara):

  https://www.fluentcpp.com

- Блог Бартломея Филипека (Bartlomiej Filipek):

  https://www.cppstories.com

- Блог Райнера Гримма (Rainer Grimm):

  http://modernescpp.com

- Блог Артура О'Двайра (Arthur O'Dwyer):

  https://quuxplusone.github.io/blog

Дополнительные ресурсы

- Сайт Бьерна Страуструпа:

  https://stroustrup.com

- Сайт организации Standard C++ Foundation:

  https://isocpp.org

- Сайт Комитета по стандартам C++:

  http://www.open-std.org/jtc1/sc22/wg21

Сайты, где ответят на ваши вопросы

К числу самых популярных в мире интернет-форумов по языку C++ (и по программированию вообще) относятся:

- https://stackoverflow.com
- https://www.reddit.com/r/cpp
- https://groups.google.com/g/comp.lang.c++
- https://www.dreamincode.net/forums/forum/15-c-and-c

Здесь вы найдете адреса еще нескольких полезных сайтов:

https://www.geeksforgeeks.org/stuck-in-programming-get-the-solution-from-these-10-best-websites

Кроме того, поставщики программного обеспечения часто создают интернет-форумы для поддержки своих инструментов и библиотек. Многие библиотеки координируются и поддерживаются на GitHub. На страницах некоторых библиотек на GitHub вы можете обращаться к их разработчикам за поддержкой через вкладку `Issues`.

Общение с авторами

Если при чтении книги у вас возникнут вопросы, вы можете связаться с нами по адресу:

deitel@deitel.com

Мы оперативно ответим.

Благодарности

Мы благодарим Барбару Дейтел (Barbara Deitel) за долгие часы, которые она посвятила поиску информации в интернете во время нашей работы над этим проектом. Нам посчастливилось работать с преданной своему делу командой про-

фессионалов издательского дела из компании Pearson. Мы высоко ценим усилия и 27-летнее наставничество нашего друга и коллеги Марка Л. Тауба (Mark L. Taub), вице-президента Pearson IT Professional Group. Марк и его команда издают наши профессиональные книги и видеопродукцию LiveLessons, а также спонсируют наши обучающие вебинары, предлагаемые через сервис O'Reilly Online Learning:

https://learning.oreilly.com

Чарви Арора (Charvi Arora) искала рецензентов и руководила процессом рецензирования. Джули Нахиль (Julie Nahil) руководила выпуском книги. Чути Прасерсит (Chuti Prasertsith) разработала дизайн обложки.

Рецензенты

В этом проекте нам посчастливилось получить рецензии на рукопись от десяти известных специалистов. Большинство рецензентов входят в состав Комитета ISO по стандартам C++, работали в нем или имеют с ним рабочие отношения. Многие из них внесли свой вклад в развитие языка. Они помогли нам сделать книгу лучше, а все оставшиеся в ней недостатки — наши собственные.

- Андреас Фертиг (Andreas Fertig), независимый преподаватель и консультант по C++, создатель сайта cppin-sights.io, автор книги «Programming with C++20».
- Марк Грегуар (Marc Gregoire), архитектор программного обеспечения, Nikon Metrology, эксперт класса Microsoft Visual C++ MVP, автор книги «Professional C++» (5-е издание, обновленное с учетом C++20).
- Д-р Дейзи Холлман (Dr. Daisy Hollman), член комитета ISO по стандартам C++.
- Дэнни Калев (Danny Kalev), доктор философии, сертифицированный системный аналитик и инженер-программист, бывший член комитета ISO по стандартам C++.
- Дитмар Куль (Dietmar Kühl), старший разработчик программного обеспечения, Bloomberg L.P., член комитета ISO по стандартам C++.
- Инбал Леви (Inbal Levi), Solar Edge Technologies, директор ISO C++ Foundation, председатель группы ISO C++ SG9 (Ranges), член комитета ISO по стандартам C++.
- Артур О'Двайр (Arthur O'Dwyer), преподаватель C++, председатель трека CppCon «Back to Basics», автор нескольких принятых предложений по развитию языка в версиях C++17, C++20 и C++23, автор книги «Mastering the C++17 STL».
- Саар Раз (Saar Raz), старший инженер-программист, Swimm.io, разработчик концептов C++20 в команде Clang.
- Хосе Антонио Гонсалес Секо (José Antonio González Seco), член парламента Андалусии.

- Энтони Уильямс (Anthony Williams), член комиссии по стандартам C++ Британского института стандартов, директор компании Just Software Solutions Ltd., автор книги «C++ Concurrency in Action» (2-е издание). Энтони является автором или соавтором многих документов Комитета по стандартам C++, которые привели к появлению стандартных средств конкурентности в C++.

Артур О'Двайр

Мы должны отметить исключительные усилия, которые Артур О'Двайр приложил к рецензированию нашей рукописи. Из его комментариев мы узнали много нового о тонкостях языка C++ и особенно о современных идиомах программирования на C++. Он не только добавил детальные комментарии во все PDF-файлы с главами книги, которые мы ему отправляли, но и предоставил нам отдельный документ с подробным объяснением своих комментариев, во многих случаях переписал код и указал на внешние ресурсы с ценной информацией. Учитывая все замечания рецензентов, мы всегда с нетерпением ждали, что скажет Артур, особенно по самым сложным вопросам. Он занятой эксперт, но щедро тратил свое время и всегда был конструктивен. Он настаивал на том, чтобы мы «сделали все правильно», и прилагал все усилия, чтобы помочь нам в этом. Артур преподает C++ профессиональным программистам. Он научил нас многим приемам правильной работы с C++.

GitHub

Благодарим GitHub за то, что он позволяет нам легко делиться своим кодом и поддерживать его в актуальном состоянии, а также за предоставление инструментов, которые позволяют десяткам миллионов разработчиков вносить свой вклад в сотни миллионов репозиториев исходного кода[1]. Эти инструменты помогают всемирному сообществу разработчиков открытого программного обеспечения, предоставляющему библиотеки для популярных языков программирования, упростить создание мощных приложений и избавить коллег от необходимости «изобретать велосипед».

Мэтт Годболт и Compiler Explorer

Благодарим Мэтта Годболта (Matt Godbolt), создателя онлайн-сервиса Compiler Explorer (https://godbolt.org), который позволяет компилировать и запускать программы на многих языках программирования. Благодаря этому сервису вы можете протестировать свой код:

- на большинстве популярных компиляторов C++, включая три рекомендуемых нами компилятора;

[1] «Where the World Builds Software». Был доступен: 15 сентября 2023 г. https://github.com/about.

- на многих выпущенных, разрабатываемых и экспериментальных версиях компиляторов.

Например, мы использовали имеющуюся в Compiler Explorer экспериментальную версию компилятора g++, чтобы продемонстрировать в главе 12 предварительную функциональность контрактов, которые надеемся увидеть в стандартной библиотеке одной из будущих версий C++. Несколько наших рецензентов использовали сайт godbolt.org, чтобы показывать нам предлагаемые изменения в коде; это помогло нам улучшить книгу.

Дитмар Куль

Благодарим Дитмара Куля (Dietmar Kühl), старшего разработчика программного обеспечения в компании Bloomberg L.P. и члена комитета ISO по стандартам C++, который поделился с нами своими взглядами на наследование, статический и динамический полиморфизм. Он помог нам точнее изложить эти темы в главах 10 и 15.

Райнер Гримм

Благодарим Райнера Гримма (Rainer Grimm), одного из самых популярных блогеров сообщества C++ (http://modernescpp.com). По мере углубления в темы C++20 поиск в Google часто приводил нас к его статьям. Райнер Гримм — профессиональный преподаватель C++, читающий курсы на немецком и английском языках. Он написал несколько книг по C++, в том числе «C++20: Get the Details», «Concurrency with Modern C++», «The C++ Standard Library» (3-е изд.) и «C++ Core Guidelines Explained». Райнер ведет блог о тех новых возможностях языка, которые ожидаются в C++23.

Брайан Гетц

Нам повезло, что рецензентом одной из наших книг — «Java How to Program (10-е издание) — был Брайан Гетц (Brian Goetz), архитектор языка Oracle Java и соавтор книги «Java Concurrency in Practice». Он предоставил нам множество идей и конструктивных комментариев, особенно по следующим темам:

- проектирование иерархии наследования (это помогло нам в работе над примерами исходного кода в главе 10 «ООП: наследование и динамический полиморфизм»);
- конкурентность в Java (это помогло нам в работе над главой 17 «Параллельные алгоритмы и конкурентность: высокоуровневый подход»).

Разработчики программ с открытым исходным кодом и блогеры

Мы благодарны технически грамотным людям со всего мира, которые участвуют в движении открытого программного обеспечения и ведут блоги, а также организа-

циям, которые помогают распространению открытого программного обеспечения и информации о нем.

Поисковая система Google

Спасибо Google, чья поисковая система отвечает на наш постоянный поток запросов, выполняя каждый из них за доли секунды, в любое время дня и ночи, причем совершенно бесплатно. Это лучший инструмент повышения производительности труда, который мы добавили в наш исследовательский процесс за последние 20 лет!

Grammarly

Теперь мы используем платную версию Grammarly для проверки всех наших рукописей. Разработчики этого сервиса отмечают, что он помогает «писать ярко, понятно, без ошибок», пользуясь «текстовым помощником на основе искусственного интеллекта»[1]. Они также утверждают: «Используя различные инновационные подходы, в том числе передовое машинное обучение и глубокое обучение, мы постоянно открываем новые возможности в области обработки естественного языка (NLP), чтобы предоставлять непревзойденную помощь»[2].

Grammarly предлагает бесплатные инструменты, которые можно интегрировать в несколько популярных веб-браузеров, а также в Microsoft® Office365™ и в Google Docs™. В дополнение к этому предлагаются более мощные премиум-инструменты и версия для бизнеса. Здесь вы можете узнать о разных вариантах бесплатного и платного использования сервиса:

https://www.grammarly.com/plans

Мы будем благодарны за ваши замечания, критику, исправления и предложения по улучшению книги, если таковые появятся у вас при чтении книги. Всю корреспонденцию, включая вопросы, мы просим направлять по адресу:

deitel@deitel.com

Мы оперативно ответим.

Мы приглашаем вас в захватывающий мир программирования на C++20. За последние 30 лет мы с удовольствием работали над 11 изданиями наших академических и профессиональных книг по C++. Мы надеемся, что книга *«C++20 для программистов. Метод готовых объектов»* вам понравится и вы получите познавательный, серьезный и интересный опыт разработки программ на современном C++.

Пол Дейтел, Харви Дейтел

---

[1] «Grammarly». Был доступен: 15 сентября 2023 г. https://www.grammarly.com.

[2] «Our Mission». Был доступен: 15 сентября 2023 г. https://www.grammarly.com/about.

Об авторах

Пол Дж. Дейтел (Paul J. Deitel), генеральный директор и технический директор компании Deitel & Associates, Inc., — выпускник Массачусетского технологического института, 42 года работающий в компьютерной сфере. Пол — один из самых опытных в мире преподавателей языков программирования, он с 1992 года ведет профессиональные курсы для разработчиков программного обеспечения. Прочитал сотни курсов по программированию для академических, промышленных, правительственных и военных организаций, которые являются клиентами Deitel & Associates. Среди них Калифорнийский университет, Cisco, IBM, Siemens, Sun Microsystems (теперь Oracle), Dell, Fidelity, NASA (Космический центр им. Кеннеди), Национальная лаборатория сильных штормов, ракетный полигон White Sands, Rogue Wave Software, Boeing, Puma, iRobot и многие другие. Пол Дж. Дейтел и его соавтор, доктор Харви М. Дейтел, являются авторами одних из самых популярных в мире учебников по языкам программирования, профессиональных книг, видеофильмов и интерактивных мультимедийных электронных курсов, а также ведущими тренингов.

Харви М. Дейтел (Harvey M. Deitel), председатель совета директоров и директор по стратегии компании Deitel & Associates, Inc., имеет более 60 лет опыта работы в области вычислительной техники. Он получил степени бакалавра и магистра электротехники в Массачусетском технологическом институте и степень доктора математики в Бостонском университете (изучал вычислительную технику по всем этим программам еще до того, как в этих вузах были созданы факультеты компьютерных наук). Харви М. Дейтел имеет большой опыт преподавания в отраслевых организациях и колледжах, он был председателем факультета компьютерных наук в Бостонском колледже, а затем вместе с сыном Полом основал в 1991 году компанию Deitel & Associates. Публикации Дейтелов получили международное признание: более 100 переводов опубликованы на японском, немецком, русском, испанском, французском, польском, итальянском, упрощенном китайском, традиционном китайском, корейском, португальском, греческом, урду и турецком языках. Доктор Дейтел прочитал сотни курсов по программированию для академических, корпоративных, правительственных и военных клиентов.

От издательства

Все примеры исходного кода, приведенные в книге, вы можете скачать в виде ZIP-файла с сайта издательства «Питер» по адресу https://clck.ru/38xMy4.



Кроме того, в качестве бонуса вы можете скачать дополнительные онлайн-главы (на английском языке), которые авторы не включили в печатное издание книги.

Авторы снабдили текст книги внушительным количеством ссылок на электронные ресурсы на английском языке, в частности на статьи из Википедии. Там, где в российской Википедии имеются аналогичные статьи, ссылки были заменены на русскоязычные.

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Перед началом работы

Перед тем как вы приступите к изучению C++, прочитайте этот раздел, чтобы узнать, что означают условные обозначения в книге и как подготовить свой компьютер к компиляции и выполнению программ.

Условные обозначения

В книге используются следующие шрифтовые обозначения.

Все примеры исходного кода, а также данные ввода-вывода набраны моноширинными шрифтами. Ключевые слова выделены **жирным**, а комментарии в коде — более светлым шрифтом. Обратите внимание, что номера в начале строк не являются частью инструкций C++ и добавлены только для наглядности:

```
1 int main() {
2 std::cout << "Welcome to C++!\n"; // Выводим сообщение
3 return 0; // Успешное завершение программы
4 }
```

В основном тексте книги моноширинным шрифтом выделены слова и символы, обозначающие идентификаторы языка C++ (в том числе имена функций, классов и объектов), операторы, значения переменных, локальные пути и имена файлов, фрагменты исходного кода, фрагменты данных ввода-вывода, названия программных компонентов и подобные компьютерные сущности.

Жирным шрифтом выделены важные термины и их английские эквиваленты.

Загрузка примеров кода

Примеры кода, включенные в книгу *«C++20 для программистов. Метод готовых объектов»*, мы храним в репозитории GitHub[1]. На веб-странице книги в разделе Source Code есть ссылка на этот репозиторий, а также прямая ссылка на ZIP-файл со всеми примерами кода:

https://deitel.com/cpp20fp

Если вы знакомы с Git и GitHub, клонируйте этот репозиторий в своей системе. Если вы скачали ZIP-файл, распакуйте его содержимое. В наших дальнейших

[1] Вы также можете скачать ZIP-файл с примерами кода с сайта издательства «Питер», ссылка приведена на с. 47 в разделе «От издательства». — *Примеч. ред.*

инструкциях мы исходим из того, что вы уже скопировали примеры кода из ZIP-файла в папку `examples`, расположенную в папке `Documents` вашей учетной записи пользователя.

Если вы еще не работали с Git и GitHub, но хотите узнать об этих важнейших инструментах разработчика, ознакомьтесь с их руководствами по адресу:

https://guides.github.com/activities/hello-world

Компиляторы, на которых мы тестировали исходный код

Убедитесь, что на вашем компьютере установлена последняя версия компилятора C++. Мы протестировали все приведенные в книге примеры кода на нескольких бесплатных компиляторах, которые рекомендуем:

- В операционной системе Microsoft Windows мы использовали интегрированную среду разработки Microsoft Visual Studio в версии Community Edition[1], включающую в себя компилятор Visual C++ и другие инструменты разработки от Microsoft.
- В операционной системе macOS мы использовали компилятор C++, входящий в состав Apple Xcode[2] и представляющий собой разновидность компилятора Clang C++.
- В операционной системе Linux мы использовали компилятор GNU C++[3], входящий в состав пакета GNU Compiler Collection (GCC). Пакет GNU C++ уже установлен на многих компьютерах с Linux, но вам, возможно, надо обновить этот компилятор до более свежей версии. Его можно устанавливать и на компьютерах с macOS и Windows.
- Вы можете запускать последние версии компиляторов GNU C++ и Clang C++ во всех трех операционных системах — Windows, macOS и Linux — с помощью контейнеров Docker.
- Далее в разделе «Перед началом работы» описана установка перечисленных компиляторов и контейнеров Docker. В главе 1 книги, в разделе 1.2, мы покажем, как с помощью этих компиляторов компилировать и запускать написанные на C++ программы.

Не все примеры кода совместимы со всеми тремя компиляторами

К моменту выхода этой книги[4] производители компиляторов еще не полностью реализовали в них некоторые из новых возможностей C++20. Они будут реали-

---

[1] Visual Studio 2022 Community Edition.

[2] Xcode 13.2.1.

[3] GNU g++ 11.2.

[4] Оригинальное издание вышло в феврале 2022 г. — *Примеч. ред.*

зованы в последующих версиях компиляторов, и тогда мы заново протестируем код, обновим наш цифровой контент и опубликуем обновления на сайте:

https://deitel.com/cpp20fp

Установка Visual Studio Community Edition в Windows

Если вы работаете в Windows, прежде всего посетите эту страницу и убедитесь, что ваш компьютер соответствует требованиям, обязательным для установки Microsoft Visual Studio Community Edition:

https://docs.microsoft.com/en-us/visualstudio/releases/2022/system-requirements

Затем перейдите по адресу загрузки:

https://visualstudio.microsoft.com/downloads

Для загрузки и установки выполните следующие шаги.

1. Нажмите кнопку `Free download` на вкладке веб-страницы `Community`.
2. Загрузите предлагаемый установочный файл (`VisualStudioSetup.exe`) и запустите его из той папки, в которую вы его загрузили.
3. В диалоговом окне `User Account Control` (`Контроль учетных записей`) нажмите кнопку `Yes`, чтобы разрешить инсталлятору внести изменения в вашу систему.
4. В диалоговом окне `Visual Studio Installer` нажмите кнопку `Continue`, чтобы разрешить инсталлятору загрузить компоненты, необходимые для дальнейшего выбора конфигурации.
5. Для работы с примерами кода из этой книги выберите в следующем окне пункт `Desktop Development with C++`, включающий компилятор Visual C++ и стандартную библиотеку C++.
6. Нажмите кнопку `Install`. Начнется процесс загрузки и установки Visual Studio, который может занять много времени, особенно если у вас относительно медленное интернет-соединение.

Установка Xcode в macOS

На компьютере с macOS выполните следующие шаги для установки Xcode.

1. Откройте меню Apple и выберите пункт `App Store...` (или нажмите на значок `App Store` в нижней панели).
2. В строке поиска `App Store` наберите: `Xcode`.
3. Нажмите кнопку `Get`, чтобы установить Xcode.

Установка актуальной версии GNU C++

Существует множество дистрибутивов Linux, в которых используются разные методы обновления программ. Обратитесь к онлайн-документации вашего дис-

трибутива, чтобы узнать, как правильно обновить GNU C++ до последней версии. Вы также можете загрузить GNU C++ для разных платформ по адресу:

https://gcc.gnu.org/install/binaries.html

Установка GNU Compiler Collection в Ubuntu Linux, запущенной в подсистеме Windows для Linux

Если вы хотите установить GNU Compiler Collection на компьютер с Windows, это можно сделать с помощью подсистемы Windows для Linux (Windows Subsystem for Linux, WSL), которая позволяет запускать Linux в Windows. Вы можете легко установить Ubuntu Linux из Microsoft Store, но сначала необходимо установить WSL.

1. В строке поиска на панели задач наберите: `Turn Windows features on or off` (`Включение или отключение компонентов Windows`), затем нажмите кнопку `Open` (`Открыть`) в результатах поиска.
2. В открывшемся окне найдите в списке пункт `Windows Subsystem for Linux` (`Подсистема Windows для Linux`). Если в нем стоит галочка, то WSL уже установлена. Если галочки нет, поставьте ее и нажмите кнопку `OK`. Windows установит WSL и предложит вам перезагрузить компьютер.
3. После перезагрузки и входа в систему откройте приложение `Microsoft Store`, выполните поиск по слову `Ubuntu`, выберите приложение `Ubuntu` и нажмите кнопку `Install` (`Установить`). После этого будет установлена последняя версия Ubuntu Linux.
4. После установки нажмите кнопку `Launch`, чтобы открыть окно командной строки Ubuntu Linux, в котором процесс установки продолжится. Вам будет предложено создать имя пользователя и пароль для Ubuntu — они не должны совпадать с вашим именем пользователя и паролем для Windows.
5. Когда установка Ubuntu завершится, выполните следующие две команды, чтобы установить пакет GCC и отладчик GNU (при этом система может попросить вас ввести пароль для учетной записи, созданной на шаге 4):

```
sudo apt-get update
sudo apt-get install build-essential gdb
```

Убедитесь, что компилятор `g++` установлен, выполнив следующую команду:

```
g++ --version
```

Перед началом работы с нашими примерами исходного кода выполните в Ubuntu команду `cd` для изменения текущей папки, например:

```
cd /mnt/c/Users/YourUserName/Documents/examples
```

Используйте в этой команде свое имя пользователя и путь к папке, в которую вы поместили файлы с примерами исходного кода.

Docker и контейнеры Docker

Docker — это утилита для упаковки программ в контейнеры (также называемые образами), в которых собрано все необходимое для выполнения этих программ на разных платформах, что особенно полезно для программных пакетов со сложными настройками и конфигурациями. Для многих таких пакетов предлагаются бесплатные готовые контейнеры Docker (обычно на сайте https://hub.docker.com), которые вы можете загрузить и запустить локально на своем компьютере. Docker — это отличный способ быстро и удобно начать работу с новыми технологиями. Это также отличный способ поэкспериментировать с новыми версиями компиляторов.

Установка Docker

Чтобы использовать контейнеры Docker, необходимо сначала установить Docker. Если вы работаете в Windows или macOS, загрузите с этого сайта и запустите программу установки Docker Desktop:

https://www.docker.com/get-started

Затем следуйте инструкциям на экране. Кроме того, зарегистрируйте на этой странице свою учетную запись Docker Hub, чтобы получить возможность пользоваться контейнерами с сайта https://hub.docker.com. Если вы работаете в Linux, установите с этого сайта Docker Engine:

https://docs.docker.com/engine/install

Загрузка контейнера Docker, содержащего GNU Compiler Collection

Команда разработчиков GNU предоставляет официальные контейнеры Docker по адресу:

https://hub.docker.com/_/gcc

После установки и запуска Docker откройте командную строку[1] (в Windows), терминал (в macOS или Linux) или командную оболочку (Linux), затем выполните команду:

```
docker pull gcc:latest
```

После этого Docker загрузит контейнер, содержащий GNU Compiler Collection (GCC) последней версии (11.2 на момент написания книги). В разделе 1.2 мы покажем, как запустить этот контейнер и использовать его для компиляции и запуска программ, написанных на C++.

---

[1] Из учетной записи обычного пользователя Windows надо открывать командную строку от имени администратора (Run as administrator).

Загрузка контейнера Docker, содержащего Clang

В настоящее время у команды разработчиков Clang нет своего официального контейнера Docker, но многие готовые к работе контейнеры есть на сайте https://hub.docker.com. Для этой книги мы использовали следующий популярный контейнер:

https://hub.docker.com/r/teeks99/clang-ubuntu

Откройте командную строку (в Windows), терминал (в macOS или Linux) или командную оболочку (в Linux), затем выполните команду:

```
docker pull teeks99/clang-ubuntu:latest
```

После этого Docker загрузит контейнер, содержащий Clang последней версии (13.0 на момент написания книги). В разделе 1.2 мы покажем, как запустить этот контейнер и использовать его для компиляции и запуска программ, написанных на C++.

Ответы на ваши вопросы по C++

Если при чтении книги у вас возникнут вопросы, с нами легко связаться по email:

deitel@deitel.com

И через форму обратной связи на сайте:

https://deitel.com/contact-us

Мы оперативно ответим.

В интернете очень много информации о программировании. Например, бесценным ресурсом как для программистов, так и для всех интересующихся является сайт Stack Overflow:

https://stackoverflow.com

На этом сайте вы можете:

- искать ответы на самые распространенные вопросы о программировании;
- искать сообщения об ошибках, чтобы узнать их причину;
- задавать вопросы о программировании, чтобы получать ответы от программистов со всего мира;
- приобретать ценные знания о программировании в целом.

Собеседников на темы C++ вы найдете на канале cpplang в мессенджере Slack:

https://cpplang-inviter.cppalliance.org

и на Discord-сервере #include<C++>:

https://www.includecpp.org/discord

Онлайн-документация по C++

На этом сайте опубликована документация к стандартной библиотеке C++:

https://cppreference.com

А здесь вы можете ознакомиться с ответами на частые вопросы по C++:

https://isocpp.org/faq

О библиотеке форматирования текста {fmt}

На протяжении всей книги мы включили во многие программы следующую строку кода:

```
#include <fmt/format.h>
```

Она позволяет нашим программам использовать средства форматирования текста, предоставляемые библиотекой с открытым исходным кодом `{fmt}`[1]. В этих программах есть вызовы функции `fmt::format`.

Новые средства форматирования текста, включенные в C++20, аналогичны той части библиотеки `{fmt}`, которая используется в наших программах. Поэтому при работе с компилятором, который уже полностью поддерживает стандарт C++20, предыдущую строку кода можно заменить следующей:

```
#include <format>
```

В этом случае надо использовать вызовы функции `std::format` (вместо аналогичных вызовов функции `fmt::format`).

Когда мы писали эту книгу, только компилятор Microsoft Visual C++ поддерживал новые средства форматирования текста, введенные в C++20. По этой причине в наших примерах используется библиотека с открытым исходным кодом `{fmt}`, гарантирующая совместимость кода с остальными рекомендуемыми компиляторами.

Средства статического анализа кода

Мы использовали три статических анализатора кода, чтобы проверить наши примеры кода на соответствие основным принципам C++, стандартам кодирования и идиомам современного C++, а также убедиться, что в них нет распространенных ошибок, возможных проблем по части безопасности, быстродействия, читаемости кода и т. д.:

- `clang-tidy` — https://clang.llvm.org/extra/clang-tidy;
- `cppcheck` — https://cppcheck.sourceforge.io;
- средства проверки кода на соответствие Microsoft C++ Core Guidelines, встроенные в статический анализатор кода Visual Studio.

---

[1] «{fmt}». Был доступен: 15 сентября 2023 г. https://github.com/fmtlib/fmt.

Вы можете установить программу `clang-tidy` в Linux следующими командами:

```
sudo apt-get update -y
sudo apt-get install -y clang-tidy
```

Вы можете установить программу `cppcheck` для различных платформ (операционных систем), следуя инструкциям на сайте https://cppcheck.sourceforge.io. В Visual C++, после того как вы выполните тест-драйв из раздела 1.2 и научитесь создавать проект, вы можете настроить статический анализатор кода таким образом:

1. Щелкните правой кнопкой мыши на имени проекта в `Solution Explorer` и выберите `Properties`.
2. В появившемся диалоговом окне выберите `Code Analysis > General` в левой колонке, а затем установите для параметра `Enable Code Analysis on Build` значение `Yes` в правой колонке.
3. Затем выберите `Code Analysis > Microsoft` в левой колонке. Теперь в правой колонке, в раскрывающемся списке, можно выбрать определенный набор правил анализа. Мы использовали опцию `<Choose multiple rule sets...>`, чтобы выбрать все наборы правил, начинающиеся с `C++ Core Check`. Нажмите кнопку `Save As...`, дайте имя своему набору правил, нажмите кнопку `Save`, а затем `Apply`. (Обратите внимание, что при этом будет выдано много предупреждений, связанных с библиотекой форматирования текста `{fmt}`, которую мы используем во многих примерах кода.)



Введение и тест-драйв популярных бесплатных компиляторов C++

В этой главе вы:

- узнаете о структуре книги и получите советы о том, с чего начать изучение большого, сложного и мощного языка C++20;
- научитесь компилировать и запускать написанное на C++ тестовое приложение с помощью трех компиляторов, которые мы рекомендуем: Visual C++ из Microsoft Visual Studio (в Windows), Clang C++ из Xcode (в macOS) и g++ из GNU Compiler Collection (в Linux);
- узнаете, как запускать компиляторы g++ и clang++ в контейнерах Docker, с которыми можно работать и в Windows, и в macOS, и в Linux;
- найдете ссылки на интернет-ресурсы, рассказывающие о более чем 40-летней истории C++;
- узнаете, почему многопоточное (конкурентное) программирование на C++ ускоряет работу приложений на компьютерах с многоядерными процессорами;
- познакомитесь с концепцией объектов, которую мы в начале книги продемонстрируем на простых примерах с готовыми объектами, а затем подробно рассмотрим в главах, посвященных объектно-ориентированному программированию (начиная с главы 9).

1.1. Введение

Добро пожаловать в C++! Это один из самых популярных в мире языков программирования[1]. Мы поможем вам изучить его в современной версии, которая называется C++20 («Си плюс плюс двадцать»). Это последняя на сегодняшний день версия, которая стандартизирована Международной организацией по стандартизации (International Organization for Standardization, ISO). Темы главы 1 рассчитаны на изучение языка практически с нуля:

- краткий обзор книги для быстрого ознакомления с ее структурой;
- тест-драйв трех популярных бесплатных компиляторов C++;
- закон Мура, многоядерные процессоры и решающая роль многопоточного программирования в создании быстродействующих приложений на C++;
- основные принципы и термины объектно-ориентированного программирования, которые будут встречаться в книге далее.

Краткий обзор этой книги

Прежде чем вы углубитесь в чтение, мы рекомендуем вам быстро взглянуть на всю структуру книги «с высоты 40 000 футов», чтобы получить общее представление о том, как вы будете изучать большой, сложный и мощный язык C++20. Для этого в книге есть:

- Оглавление в виде наглядной схемы на внутренней стороне обложки.
- Предисловие, поясняющее суть книги и наш подход к современному программированию на C++. Здесь мы представляем метод готовых объектов, который позволит вам использовать минимум простых инструкций C++, чтобы заставить мощные классы выполнять важные задачи. Вы сможете это делать задолго до того, как самостоятельно напишете ваши первые классы. Обязательно прочтите в предисловии раздел «Краткое содержание книги», в котором указаны ключевые темы каждой главы. При чтении этого раздела вам будет удобно сверяться со схемой-оглавлением.

Ресурсы по истории C++

В 1979 году Бьерн Страуструп (Bjarne Stroustrup; в другой транскрипции — Бьярне Страуструп) приступил к разработке языка программирования C++, который он сначала назвал «C with Classes»[2]. Сегодня не менее 5 миллионов разработчиков (по некоторым оценкам — до 7,5 миллиона[3, 4]) используют C++ для создания

---

[1] «TIOBE Index». Был доступен: 15 сентября 2023 г. https://www.tiobe.com/tiobe-index.

[2] «Страуструп, Бьерн». https://ru.wikipedia.org/wiki/Страуструп,_Бьерн.

[3] State of the Developer Nation, 21st Edition», Q3 2021. Был доступен: 15 сентября 2023 г. https://www.slashdata.co/free-resources/state-of-the-developer-nation-21st-edition.

[4] Тим Андерсон (Tim Anderson), «Report: World's Population of Developers Expands, JavaScript Reigns, C# Overtakes PHP», 26 апреля 2021 г. Был доступен: 15 сентября 2023 г. https://www.theregister.com/2021/04/26/report_developers_slashdata.

критически важных цифровых систем и прикладных программ (приложений)[1, 2]. На этом языке написаны многие компоненты операционных систем Windows[3] и macOS[4]. C++ широко применяется при разработке браузеров (например, Google Chrome[5] и Mozilla Firefox[6]), систем управления базами данных (например, MySQL[7] и MongoDB[8]) и многих других видов приложений.

История языка C++ хорошо документирована:

- Эта статья в «Википедии» содержит подробную историю C++ и много полезных ссылок:

  https://ru.wikipedia.org/wiki/C++

- Бьерн Страуструп, создатель C++, рассказывает о том, как устроен этот язык и как он совершенствовался с момента создания до появления современной версии C++20:

  https://www.stroustrup.com/C++.html#design

- Здесь вы найдете список основных этапов развития C++ с момента создания и другие исторические сведения:

  https://ru.cppreference.com/w/cpp/language/history

1.2. Тест-драйв: компиляция приложения, написанного на C++20

В этом разделе вы будете компилировать, запускать и проверять в работе ваше первое приложение на C++[9]. Это игра, в которой надо угадать случайное число от 1 до 1000. Если вы введете правильное число, игра закончится. Если вы ошибетесь, приложение сообщит, что вы ввели слишком большое или слишком маленькое число. Нет никаких ограничений на количество попыток, которые вы можете сделать.

---

1 «Top 10 Reasons to Learn C++». Был доступен: 15 сентября 2023 г. https://www.geeksforgeeks.org/top-10-reasons-to-learn-c-plus-plus.

2 «What Is C++ Used For? Top 12 Real-World Applications and Uses of C++». Был доступен: 15 сентября 2023 г. https://www.softwaretestinghelp.com/cpp-applications.

3 «What Programming Language Is Windows Written In?» Был доступен: 15 сентября 2023 г. https://social.microsoft.com/Forums/en-US/65a1fe05-9c1d-48bf-bd40-148e6b3da9f1/what-programming-language-is-windows-written-in.

4 «macOS». https://ru.wikipedia.org/wiki/MacOS.

5 «Google Chrome». https://ru.wikipedia.org/wiki/Google_Chrome.

6 «Firefox». https://ru.wikipedia.org/wiki/Firefox.

7 «MySQL». https://ru.wikipedia.org/wiki/MySQL.

8 «MongoDB». https://ru.wikipedia.org/wiki/MongoDB.

9 Мы намеренно не приводим исходный код этого приложения. Пока мы хотим только показать вам, как компилировать и запускать программы с помощью трех компиляторов, которые мы рекомендуем. А с генерацией случайных чисел вы познакомитесь в главе 5.

Рассматриваемые компиляторы и среды разработки

Мы покажем, как компилировать исходный код на языке C++ и запускать скомпилированную программу в трех операционных системах:

- в Windows — с помощью интегрированной среды разработки Microsoft Visual Studio 2022 Community Edition (раздел 1.2.1);
- в macOS — с помощью интегрированной среды разработки Apple Xcode (раздел 1.2.2);
- в Linux — с помощью компилятора g++, который входит в набор GNU Compiler Collection (раздел 1.2.3);
- в любой из трех операционных систем — с помощью этого же компилятора g++, работающего в контейнере Docker (раздел 1.2.4);
- в любой из трех операционных систем — с помощью компилятора clang++, работающего в контейнере Docker (раздел 1.2.5).

Вы можете прочитать только тот раздел, который соответствует вашему компьютеру. Чтобы использовать компиляторы g++ и clang++ в контейнерах Docker, предварительно установите и запустите Docker, как описано в разделе «Перед началом работы».

1.2.1. Компиляция и запуск приложения в Windows с помощью Visual Studio 2022

В этом разделе вы скомпилируете и запустите написанную на C++ программу в операционной системе Windows, используя интегрированную среду разработки Microsoft Visual Studio 2022 Community Edition[1]. Она существует в нескольких версиях. Параметры, меню и необходимые действия, описанные в этом разделе, могут незначительно отличаться в зависимости от версии. Далее мы будем использовать термины Visual Studio и IDE (Integrated Development Environment — интегрированная среда разработки).

Шаг 1. Подготовка программного обеспечения

Если вы еще не сделали этого, прочитайте раздел «Перед началом работы», установите Visual Studio и скачайте с сайта Deitel файл с включенными в книгу примерами исходного кода.

Шаг 2. Запуск Visual Studio

Откройте Visual Studio из меню операционной системы `Start` (`Пуск`). Закройте начальное окно Visual Studio, нажав клавишу `Esc`. Не щелкайте мышью на кнопке `X` в правом верхнем углу: она завершает работу Visual Studio. Вы сможете снова открыть начальное окно в любое время, выбрав `File > Start Window`. Мы будем

[1] Мы рассматриваем Microsoft Visual Studio 2022 Community Edition версии 17.0.5.

обозначать символом > выбор пункта меню. Например, File > Open означает «выберите пункт Open в меню File».

Шаг 3. Создание проекта

Проект (project) — это группа взаимосвязанных файлов (содержащих исходный код на языке C++), из которых состоит приложение. Visual Studio позволяет объединять разрабатываемые приложения в проекты и **решения (solutions)**. Каждое решение может содержать один или несколько проектов. Решения, состоящие из нескольких проектов, необходимы только для очень больших приложений. Для каждого приложения из этой книги нужно решение только с одним проектом. Изучая наши примеры кода, вы будете начинать с **пустого проекта (empty project)** и добавлять в него файлы. Для создания проекта:

1. Выберите File > New > Project..., чтобы открыть диалоговое окно Create a New Project.
2. Выберите шаблон с именем Empty Project и тегами C++, Windows и Console. Он предназначен для программ, выполняемых в командной строке. (В зависимости от версии и настроек Visual Studio в списке может быть много других шаблонов. Для поиска нужного шаблона введите его имя в строку Search for templates). Щелкните на кнопке Next, чтобы открыть диалоговое окно Configure your new project.
3. Укажите Project name (имя проекта) и Location (путь к файлам проекта). В качестве имени мы выбрали cpp20_test. В качестве пути мы выбрали папку examples, содержащую примеры кода из этой книги. Щелкните на кнопке Create, чтобы создать новый проект.

Visual Studio создаст ваш проект, назначив для его файлов папку:

```
C:\Users\ИмяПользователя\Documents\examples
```

(или другую указанную вами папку), и откроет главное окно.

Когда вы пишете и редактируете код, Visual Studio отображает каждый файл с кодом вашего приложения в отдельной вкладке. Для просмотра этих файлов и для управления ими предназначено окно Solution Explorer (оно может находиться справа или слева в главном окне Visual Studio). В нем есть папка Source Files, в которую вы будете помещать файлы с нашими примерами кода. Если Solution Explorer не отображается, вы можете вывести его на экран, выбрав View > Solution Explorer.

Шаг 4. Добавление файла GuessNumber.cpp в проект

Теперь добавьте файл GuessNumber.cpp к созданному на шаге 3 проекту. Для этого в Solution Explorer:

1. Щелкните правой кнопкой мыши на папке Source Files и выберите Add > Existing Item....

2. В появившемся диалоговом окне перейдите в папку `ch01`, вложенную в папку `examples`, выберите `GuessNumber.cpp` и щелкните на кнопке `Add`[1].

Шаг 5. Настройка проекта для работы с C++20

Входящий в состав Visual Studio компилятор C++ поддерживает несколько версий стандарта C++. Для работы с этой книгой надо выбрать в параметрах проекта версию C++20:

1. В `Solution Explorer` щелкните правой кнопкой мыши на узле проекта **cpp20_test** и выберите `Properties`, чтобы открыть диалоговое окно `Property Pages` проекта `cpp20_test`.
2. В раскрывающемся списке `Configuration` выберите `All Configurations`. В раскрывающемся списке `Platform` выберите `All Platforms`.
3. В левом столбце разверните узел `C/C++` и выберите `Language`.
4. В правом столбце щелкните в поле справа от `C++ Language Standard`, затем щелкните на стрелке вниз, выберите `ISO C++20 Standard (/std:c++20)` и щелкните на кнопке `OK`.

Шаг 6. Компиляция и запуск проекта

Чтобы скомпилировать и запустить проект, то есть проверить приложение в работе, выберите `Debug > Start without debugging` или нажмите `Ctrl + F5`. Если исходный код компилируется без ошибок, Visual Studio откроет окно командной строки и выполнит приложение. Вы увидите такое сообщение:

```
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?
```

Шаг 7. Ввод первого числа

После знака `?` в командной строке введите `500` и нажмите `Enter`. Ответ приложения будет меняться при каждом его запуске. У нас приложение вывело сообщение `"Too low. Try again."`, указывающее, что введенное число меньше случайного числа, которое надо угадать:

```
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too low. Try again.
?
```

[1] Если исходный код состоит из нескольких файлов (вы встретите такие программы в последующих главах), выбирайте в этом диалоговом окне все эти файлы сразу. Когда вы начнете писать программы самостоятельно, вы можете щелкнуть правой кнопкой мыши на папке Source Files и выбрать Add > New Item..., чтобы открыть диалоговое окно добавления нового файла.

Шаг 8. Ввод второго числа

После следующего знака ?, если первое число оказалось слишком маленьким, введите 750 и нажмите Enter; в противном случае введите 250 и нажмите Enter. Например, мы ввели 750, и приложение вывело сообщение "Too high. Try again.", потому что введенное значение оказалось слишком большим:

```
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too low. Try again.
? 750
Too high. Try again.
?
```

Шаг 9. Ввод следующих чисел

Продолжайте игру, вводя разные числа, пока не угадаете то число, которое выбрано приложением. После этого вы увидите сообщение "Excellent! You guessed the number!":

```
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too low. Try again.
? 750
Too high. Try again.
? 625
Too high. Try again.
? 562
Too low. Try again.
? 593
Too low. Try again.
? 607
Too low. Try again.
? 616
Too high. Try again.
? 612
Too low. Try again.
? 614
Too high. Try again.
? 613

Excellent! You guessed the number!
Would you like to play again (y or n)?
```

Шаг 10. Повторная игра или завершение работы приложения

Когда число будет угадано, приложение спросит, не хотите ли вы сыграть еще раз: "Would you like to play again (y or n)?". При нажатии клавиши y приложение вы-

берет другое случайное число и игра начнется сначала. При нажатии клавиши n приложение завершит работу.

Повторное использование проекта для последующих примеров

Вы можете создавать отдельный проект для каждого приложения из книги, выполняя описанные в этом разделе действия. Или, возможно, вам будет удобнее работать с одним проектом, каждый раз удаляя из него файл с исходным кодом и добавляя вместо него следующий файл. Чтобы удалить файл из вашего проекта (но не из вашей системы), выберите этот файл в `Solution Explorer`, затем нажмите клавишу `Del` (`Delete`). Теперь вы можете повторить шаг 4, чтобы добавить другой файл в этот же проект.

Работа с Ubuntu в подсистеме Windows для Linux

Некоторые пользователи Windows могут захотеть работать на своих Windows-компьютерах с компилятором g++, входящим в пакет GNU Compiler Collection (GCC). Это можно сделать двумя способами: использовать контейнер Docker, содержащий GCC (раздел 1.2.4), или запускать этот компилятор в операционной системе Ubuntu, установленной в подсистеме Windows для Linux. Чтобы установить подсистему Windows для Linux, следуйте инструкциям по ссылке:

https://docs.microsoft.com/ru-ru/windows/wsl/install

Установите в подсистеме Windows для Linux приложение Ubuntu (см. раздел «Перед началом работы»). Чтобы перейти в папку с файлом исходного кода приложения, введите в командной строке Ubuntu следующую команду:

```
cd /mnt/c/Users/ИмяПользователя/Documents/examples/ch01
```

и продолжите с шага 2 из раздела 1.2.3 («Компиляция и запуск приложения в Linux с помощью g++»).

1.2.2. Компиляция и запуск приложения в macOS с помощью Xcode

В этом разделе вы скомпилируете и запустите написанную на C++ программу в операционной системе macOS, используя интегрированную среду разработки Apple Xcode[1].

Шаг 1. Подготовка программного обеспечения

Если вы еще не сделали этого, прочтите раздел «Перед началом работы», установите Xcode и скачайте с сайта Deitel все наши примеры исходного кода.

Шаг 2. Запуск Xcode

Откройте окно `Finder`, выберите `Applications` (`Программы`) и дважды щелкните на значке Xcode.



[1] Мы рассматриваем Xcode версии 13.2.1.

Если вы впервые запускаете Xcode, появится окно `Welcome to Xcode`. Закройте это окно. Вы сможете снова открыть его в любое время, выбрав `Window > Welcome to Xcode`. Мы будем обозначать символом > выбор пункта меню. Например, `File > Open` означает «выберите пункт `Open` в меню `File`».

Шаг 3. Создание проекта

Проект (project) — это группа взаимосвязанных файлов (содержащих исходный код на языке C++), из которых состоит приложение. Все проекты Xcode, которые мы создали для этой книги, представляют собой программы командной строки, которые вы будете выполнять непосредственно в IDE. Для создания проекта:

1. Выберите `File > New > Project...`.
2. В верхней части диалогового окна `Choose a template for your new project` щелкните на `macOS`.
3. В разделе `Application` выберите `Command Line Tool` и щелкните на кнопке `Next`.
4. В поле `Product Name` введите имя вашего проекта. В качестве имени мы указали `cpp20_test`.
5. В раскрывающемся списке `Language` выберите `C++` и щелкните на кнопке `Next`.
6. Укажите, где вы хотите сохранить проект. Мы выбрали папку `examples`, содержащую примеры из этой книги.
7. Щелкните на `Create`.

После создания проекта откроется главное окно Xcode, содержащее три области: слева — `Navigator`, в центре — `Editor` и справа — `Utilities`.

В верхней части `Navigator` находятся значки переключения панелей. Для работы с этой книгой вам будут нужны в основном две панели:

`Project` () — показывает все файлы и папки вашего проекта.

`Issue` () — показывает выдаваемые компилятором предупреждения и сообщения об ошибках.

При нажатии на значок отображается соответствующая панель.

Центральная область `Editor` предназначена для редактирования исходного кода и настройки параметров проекта. Она всегда присутствует в главном окне Xcode. Когда вы выбираете файл в панели `Project`, его содержимое отображается в `Editor`. Справа, в `Utilities`, обычно отображаются **инспекторы (inspectors)**. Например, если вы будете писать приложение для iPhone, содержащее сенсорную кнопку, то сможете настроить в этой области свойства кнопки (ее заголовок, размер, расположение и т. д.). При работе с этой книгой вы не будете использовать `Utilities`. Существует также область отладки `Debug`, в которой вы будете взаимодействовать с запущенной программой. Она появляется под областью `Editor`.

На верхней панели инструментов есть кнопки запуска и остановки программы, индикатор процесса выполнения задач, кнопки сокрытия и отображения левой (`Navigator`) и правой (`Utilities`) областей.

Шаг 4. Настройка проекта для работы с C++20

Входящий в состав Xcode компилятор Apple Clang поддерживает несколько версий стандарта C++. Для работы с этой книгой надо выбрать в параметрах проекта версию C++20:

1. В панели `Project` выберите название вашего проекта (`cpp20_test`).
2. В левой части `Editor` выберите название вашего проекта в разделе `TARGETS`.
3. В верхней части `Editor` выберите `Build Settings`, а чуть ниже нажмите `All`.
4. Перейдите к разделу `Apple Clang > Language > C++`.
5. Щелкните на значении справа от `C++ Language Dialect` и выберите `GNU++20 [-std=gnu++20]`.
6. Щелкните на значении справа от `C++ Standard Library` и выберите `Compiler Default`.

Шаг 5. Удаление файла main.cpp из проекта

По умолчанию Xcode создает файл `main.cpp`, содержащий исходный код простой программы, которая выводит строку `"Hello, World!"`. В этом тест-драйве вам не понадобится файл `main.cpp`, поэтому удалите его. В панели `Project` сделайте «правый щелчок» (`Ctrl` + щелчок) на файле `main.cpp` и выберите `Delete`. В появившемся диалоговом окне нажмите `Move to Trash`. Файл не будет удален из вашей системы до тех пор, пока вы не очистите корзину.

Шаг 6. Добавление файла GuessNumber.cpp в проект

В окне `Finder` откройте папку `ch01` из папки `examples`, содержащей наши примеры кода, и перетащите файл `GuessNumber.cpp` в папку `cpp20_test` на панели `Project`. В появившемся диалоговом окне убедитесь, что установлен флаг `Copy items if needed`, и нажмите кнопку `Finish`[1].

Шаг 7. Компиляция и запуск проекта

Чтобы скомпилировать и запустить проект, то есть проверить приложение в работе, нажмите кнопку запуска (▶) на панели инструментов Xcode. Если исходный код компилируется без ошибок, Xcode откроет область `Debug` и выполнит в ее правой половине приложение, которое выведет строку `"Please type your first guess."` и вопросительный знак (`?`) в качестве запроса данных:

```
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?
```

[1] Если исходный код состоит из нескольких файлов (вы встретите такие программы в последующих главах), перетащите все эти файлы в папку проекта. Когда вы начнете писать программы самостоятельно, вы можете сделать «правый щелчок» на папке проекта и выбрать New File..., чтобы открыть диалоговое окно добавления нового файла.

Шаг 8. Ввод первого числа

Щелкните мышью в области Debug, введите 500 и нажмите Return. Ответ приложения будет меняться при каждом его запуске. У нас приложение вывело сообщение "Too high. Try again.", указывающее, что введенное число больше случайного числа, которое надо угадать:

```
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
?
```

Шаг 9. Ввод второго числа

После следующего знака ?, если первое число оказалось слишком маленьким, введите 750 и нажмите Return; в противном случае введите 250 и нажмите Return. Например, мы ввели 250, и приложение вывело сообщение "Too high. Try again.", потому что введенное значение оказалось слишком большим:

```
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
? 250
Too high. Try again.
?
```

Шаг 10. Ввод следующих чисел

Продолжайте игру, вводя разные числа, пока не угадаете то число, которое выбрано приложением. После этого вы увидите сообщение "Excellent! You guessed the number!":

```
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
? 250
Too high. Try again.
? 125
Too low. Try again.
? 187
Too high. Try again.
? 156
Too high. Try again.
? 140
Too low. Try again.
```

```
? 148
Too high. Try again.
? 144
Too high. Try again.
? 142
Too low. Try again.
? 143

Excellent! You guessed the number!
Would you like to play again (y or n)?
```

Шаг 11. Повторная игра или завершение работы приложения

Когда число будет угадано, приложение спросит, не хотите ли вы сыграть еще раз: `"Would you like to play again (y or n)?"`. При нажатии клавиши `y` приложение выберет другое случайное число и игра начнется сначала. При нажатии клавиши `n` приложение завершит работу.

Повторное использование проекта для последующих примеров

Вы можете создавать отдельный проект для каждого приложения из книги, выполняя описанные в этом разделе действия. Или, возможно, вам будет удобнее работать с одним проектом, каждый раз удаляя из него файл с исходным кодом и добавляя вместо него следующий файл. Чтобы удалить файл из вашего проекта (но не из вашей системы), сделайте «правый щелчок» (`Ctrl` + щелчок) на имени файла в панели `Project` и выберите `Delete`. В появившемся диалоговом окне нажмите `Remove Reference`. Теперь вы можете повторить шаг 6, чтобы добавить другой файл в этот же проект.

1.2.3. Компиляция и запуск приложения в Linux с помощью g++

В этом разделе вы скомпилируете и запустите написанную на C++ программу в командной оболочке Linux, используя компилятор g++ из пакета GNU Compiler Collection (GCC)[1]. Мы предполагаем, что вы уже прочитали раздел «Перед началом работы» и поместили все наши примеры исходного кода в каталог `Documents/examples`, расположенный в каталоге вашей учетной записи пользователя.

Шаг 1. Переход в каталог ch01

Выполните команду `cd` в командной оболочке Linux, чтобы перейти в каталог `ch01` (первый пример из каталога `examples`):

```
~$ cd ~/Documents/examples/ch01
~/Documents/examples/ch01$
```

[1] Мы рассматриваем g++ версии 11.2. Чтобы узнать номер вашей версии компилятора g++, выполните команду g++ --version. Если у вас более старая версия g++, найдите в интернете инструкции по обновлению GNU Compiler Collection (GCC) для вашей системы Linux или используйте контейнер Docker с GCC, описанный в разделе 1.2.4.

В этом разделе мы будем выделять в командах жирным шрифтом те данные, которые вводит пользователь. **Приглашение (prompt)** в оболочке Ubuntu Linux обозначает тильдой (~) домашний каталог. Приглашение обычно заканчивается знаком доллара (`$`), но может отличаться в зависимости от вашей системы Linux.

Шаг 2. Компиляция приложения

Перед запуском приложения надо скомпилировать его следующей командой компилятора g++[1]:

```
~/Documents/examples/ch01$ g++ -std=c++20 GuessNumber.cpp -o GuessNumber
~/Documents/examples/ch01$
```

В этой команде:

- Параметр `-std=c++20` указывает компилятору, что мы используем C++20.
- Парамстр `-o` присваивает исполняемому файлу имя (`GuessNumber`), которое затем будет нужно для запуска программы. Если вы не включите этот параметр, g++ автоматически присвоит исполняемому файлу имя `a.out`.

Шаг 3. Запуск приложения

Введите `./GuessNumber` в командной строке и нажмите `Enter`, чтобы запустить программу:

```
~/Documents/examples/ch01$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?
```

Символы `./` перед `GuessNumber` указывают системе Linux, что надо запустить программу `GuessNumber` из текущего каталога.

Шаг 4. Ввод первого числа

Приложение выведет сообщение `"Please type your first guess."` и вопросительный знак (`?`) в качестве запроса данных. Введите `500` (ответ приложения будет меняться при каждом его запуске):

```
~/Documents/examples/ch01$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
?
```

[1] Если на вашем компьютере установлено несколько версий g++, то нужную версию можно запустить командой g++-##, где ## — номер версии. Например, команда g++-11 запускает последнюю на сегодняшний день версию 11.x.

У нас приложение вывело сообщение "Too high. Try again.", указывающее, что введенное число больше случайного числа, которое надо угадать.

Шаг 5. Ввод второго числа

После следующего знака ?, если первое число оказалось слишком маленьким, введите 750 и нажмите Enter; в противном случае введите 250 и нажмите Enter. Например, мы ввели 250, и приложение вывело строку "Too high. Try again.", потому что введенное значение оказалось слишком большим:

```
~/Documents/examples/ch01$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
? 250
Too high. Try again.
?
```

Шаг 6. Ввод следующих чисел

Продолжайте игру, вводя разные числа, пока не угадаете то число, которое выбрано приложением. После этого вы увидите сообщение "Excellent! You guessed the number!":

```
? 125
Too high. Try again.
? 62
Too low. Try again.
? 93
Too low. Try again.
? 109
Too high. Try again.
? 101
Too low. Try again.
? 105
Too high. Try again.
? 103
Too high. Try again.
? 102

Excellent! You guessed the number!
Would you like to play again (y or n)?
```

Шаг 7. Повторная игра или завершение работы приложения

Когда число будет угадано, приложение спросит, не хотите ли вы сыграть еще раз: "Would you like to play again (y or n)?". При нажатии клавиши y приложение выберет другое случайное число и игра начнется сначала. При нажатии клавиши n приложение завершит работу.

1.2.4. Компиляция и запуск приложения в контейнере Docker с помощью g++

С компилятором g++, входящим в состав GNU Compiler Collection (GCC), вы можете работать не только в Linux, но также в Windows и macOS. Один из лучших способов реализовать это кросс-платформенное решение — запустить **контейнер (container)** Docker, содержащий GCC. Мы предполагаем, что вы уже установили Docker Desktop (в Windows или macOS) или Docker Engine (в Linux) по инструкции из раздела «Перед началом работы».

Запуск контейнера Docker с GNU Compiler Collection (GCC)

Откройте командную строку (в Windows), терминал (в macOS/Linux) или командную оболочку (в Linux). Затем выполните следующие действия, чтобы запустить контейнер Docker с GCC:

1. С помощью команды `cd` перейдите в папку `examples`, содержащую все наши примеры исходного кода.
2. В Windows: запустите контейнер Docker следующей командой[1]:

```
docker run --rm -it -v "%CD%":/usr/src gcc:latest
```

3. В macOS/Linux: запустите контейнер Docker следующей командой:

```
docker run --rm -it -v "$(pwd)":/usr/src gcc:latest
```

В этих командах:

- `--rm` очищает ресурсы контейнера, когда вы его закрываете.
- `--it` запускает контейнер в интерактивном режиме, поэтому вы можете вводить команды для изменения папок, а также для компиляции и запуска программ с помощью g++.
- `-v "%CD%":/usr/src` (в Windows) или `-v "$(pwd)":/usr/src` (в macOS/Linux) позволяет контейнеру Docker получить доступ к файлам в папке, из которой выполняется команда `docker run`. В контейнере Docker вы перейдете с помощью команды `cd` к папке `/usr/src`, чтобы компилировать и запускать примеры из книги. Другими словами, ваша локальная системная папка будет сопоставлена с папкой `/usr/src` в контейнере Docker.
- `gcc:latest` — это имя контейнера. Параметр `:latest` указывает, что вы хотите использовать новейшую версию контейнера `gcc`[2].

Когда контейнер будет запущен, вы увидите примерно такое приглашение:

```
root@67773f59d9ea:/#
```

---

[1] Может появиться уведомление с просьбой разрешить контейнеру Docker доступ к файлам в текущей папке. Вам надо дать это разрешение, иначе вы не сможете получить из Docker доступ к нашим файлам исходного кода.

[2] Если вы хотите обновить ваш контейнер до последней версии, выполните команду docker pull gcc:latest перед запуском контейнера.

В контейнере работает операционная система Linux. В приглашении контейнера имя папки, в которой вы находитесь, отображается между символами : и #.

Переход к папке ch01 в контейнере Docker

Введенная вами команда `docker run` присоединила вашу папку `examples` к папке `/usr/src` файловой системы контейнера. Теперь в контейнере Docker с помощью команды `cd` перейдите к папке `ch01`, которая находится в `/usr/src`:

```
root@01b4d47cadc6:/# cd /usr/src/ch01
root@01b4d47cadc6:/usr/src/ch01#
```

Для компиляции, запуска и тестирования приложения `GuessNumber` в контейнере Docker выполните шаги 2–7 раздела 1.2.3 («Компиляция и запуск приложения в Linux с помощью g++»).

Завершение работы контейнера Docker

Чтобы завершить работу контейнера, нажмите `Ctrl` + `d` в его командной строке.

1.2.5. Компиляция и запуск приложения в контейнере Docker с помощью clang++

Вы можете работать в разных операционных системах не только с g++, но и с компилятором LLVM/Clang C++ (clang++), который тоже предназначен для командной строки. В настоящее время разработчики LLVM / Clang не предоставляют свой официальный контейнер Docker, но многие готовые к работе контейнеры есть на сайте hub.docker.com. Мы предполагаем, что вы уже установили Docker Desktop (в Windows или macOS) или Docker Engine (в Linux) по инструкции из раздела «Перед началом работы».

Мы используем актуальную и самую распространенную 13-ю версию clang++, которую вы можете загрузить с помощью следующей команды[1]:

```
docker pull teeks99/clang-ubuntu:13
```

Запуск контейнера Docker с образом teeks99/clang-ubuntu

Откройте командную строку (Windows), терминал (macOS/Linux) или командную оболочку (Linux) и выполните следующие действия, чтобы запустить контейнер Docker, содержащий образ `teeks99/clang-ubuntu`.

1. С помощью команды `cd` перейдите в папку `examples`, содержащую все наши примеры исходного кода.

[1] В Xcode включена не самая новая версия компилятора Clang C++, и в ней реализовано не так много возможностей C++20, как в версии от команды LLVM/Clang. Кроме того, на момент написания этой книги команда docker pull с параметром «latest» вместо «13» загружает контейнер Docker с 12-й, а не 13-й версией clang++.

2. В Windows: запустите контейнер Docker следующей командой[1]:

```
docker run --rm -it -v "%CD%":/usr/src teeks99/clang-ubuntu:13
```

3. В macOS / Linux: запустите контейнер Docker следующей командой:

```
docker run --rm -it -v "$(pwd)":/usr/src teeks99/clang-ubuntu:13
```

В этих командах:

- `--rm` очищает ресурсы контейнера, когда вы его закрываете.
- `--it` запускает контейнер в интерактивном режиме, поэтому вы можете вводить команды для изменения папок, а также для компиляции и запуска программ с помощью clang++.
- `-v "%CD%":/usr/src` (Windows) или `-v "$(pwd)":/usr/src` (macOS/Linux) позволяет контейнеру Docker получить доступ к файлам в папке, из которой выполняется команда `docker run`. В контейнере Docker вы перейдете с помощью команды `cd` к папке `/usr/src`, чтобы компилировать и запускать примеры из книги. Другими словами, ваша локальная системная папка будет сопоставлена с папкой `/usr/src` в контейнере Docker.
- `teeks99/clang-ubuntu:13` — это имя контейнера.

Когда контейнер будет запущен, вы увидите примерно такое приглашение:

```
root@9753bace2e87:/#
```

В контейнере работает операционная система Linux. В приглашении контейнера имя папки, в которой вы находитесь, отображается между символами : и #.

Переход к папке ch01 в контейнере Docker

Введенная вами команда `docker run` присоединила вашу папку `examples` к папке `/usr/src` файловой системы контейнера. Теперь в контейнере Docker с помощью команды `cd` перейдите к папке `ch01`, которая находится в `/usr/src`:

```
root@9753bace2e87:/# cd /usr/src/ch01
root@9753bace2e87:/usr/src/ch01#
```

Компиляция приложения

Перед запуском приложения надо его скомпилировать. Например, если в контейнере установлен компилятор `clang++-13`, выполните следующую команду:

```
clang++-13 -std=c++20 GuessNumber.cpp -o GuessNumber
```

В этой команде:

- Параметр `-std=c++20` указывает компилятору, что мы используем C++20.

[1] Может появиться уведомление с просьбой разрешить контейнеру Docker доступ к файлам в текущей папке. Вам надо дать это разрешение, иначе вы не сможете получить доступ к нашим файлам исходного кода из Docker.

- Параметр `-o` присваивает исполняемому файлу имя (`GuessNumber`), которое затем будет нужно для запуска программы. Если вы не включите этот параметр, clang++ автоматически присвоит исполняемому файлу имя `a.out`.

Запуск приложения

Для запуска и тестирования приложения `GuessNumber` в контейнере Docker выполните шаги 3–7 из раздела 1.2.3 («Компиляция и запуск приложения в Linux с помощью g++»).

Завершение работы контейнера Docker

Чтобы завершить работу контейнера, нажмите `Ctrl + d` в его командной строке.

1.3. Закон Мура, многоядерные процессоры и многопоточное программирование

Многие современные персональные компьютеры могут выполнять миллиарды вычислительных операций в секунду — больше, чем человек способен выполнить за всю жизнь. Суперкомпьютеры уже выполняют тысячи триллионов (квадриллионы) операций в секунду. Японский суперкомпьютер Fugaku может выполнять более 442 квадриллионов операций в секунду (442.01 петафлопс)[1]. Чтобы вы представили себе этот масштаб, поясним, что Fugaku может за одну секунду выполнить около 40 миллионов вычислений для каждого человека на планете! И пределы возможного для суперкомпьютеров быстро растут.

Закон Мура

Вы привыкли, что многие товары и услуги каждый год немного дорожают. Но в сфере компьютеров и связи все наоборот, особенно если говорить об аппаратном обеспечении (hardware). За много лет оно стало значительно доступнее.

На протяжении десятилетий вычислительная мощность компьютеров удваивалась примерно каждые два года. Это впечатляющее явление часто называют законом Мура — в честь Гордона Мура, соучредителя корпорации Intel, указавшего на эту тенденцию еще в 1960-х. Intel является ведущим производителем процессоров для компьютеров и встраиваемых систем (так называются электронные компоненты умной бытовой техники, домашних систем безопасности, роботов, интеллектуальных транспортных развязок и т. д.). Закон Мура и другие похожие тенденции больше всего проявляются в следующих характеристиках компьютеров:

- объемы памяти (оперативной памяти, в которой выполняются программы и обрабатываются данные);
- объемы накопителей (HDD, SSD и других дисков, на которых хранятся программы и данные);

---

[1] «Top500». Был доступен: 15 сентября 2023 г. https://ru.wikipedia.org/wiki/Top500.

- тактовая частота процессоров (количество инструкций, последовательно выполняемых в единицу времени).

Топ-менеджеры компаний NVIDIA и Arm, производящих процессоры, считают, что закон Мура больше не действует[1, 2]. Точнее, вычислительная мощность компьютеров по-прежнему растет, но теперь уже в результате новых конструктивных решений, таких как многоядерные процессоры.

Многоядерные процессоры и быстродействие

Почти в каждом современном компьютере есть **многоядерный процессор (multi-core processor)**, экономично объединяющий несколько вычислительных блоков (ядер) на одном кристалле большой микросхемы. Например, 2-ядерный процессор содержит два вычислительных блока, 4-ядерный — четыре, 8-ядерный — восемь. В основном тестовом компьютере авторов книги установлен 8-ядерный процессор Intel. Новые процессоры Apple M1 Pro и M1 Max имеют 10 ядер, в дополнение к которым в топовом процессоре M1 Pro есть 16-ядерный видеопроцессор, а в топовом процессоре M1 Max — 32-ядерный; кроме того, в каждом из них есть 16 ядер системы «Neural Engine», предназначенной для машинного обучения[3, 4]. Компания Intel выпускает 72-ядерные процессоры[5] и проектирует 80-ядерные[6]. Компания AMD проектирует модели, имеющие 192 и 256 ядер[7]. Количество ядер будет продолжать расти.

В многоядерной системе несколько вычислительных блоков могут одновременно выполнять разные части текущей задачи, ускоряя работу вашей программы. Чтобы воспользоваться всеми преимуществами многоядерной архитектуры, вам надо научиться писать многопоточные приложения. Когда приложение распределяет задачи между отдельными потоками, многоядерная система может запускать эти потоки параллельно, если доступно достаточное количество ядер.

---

1 «Moore's Law Turns 55: Is It Still Relevant?». Был доступен: 15 сентября 2023 г. https://www.techrepublic.com/article/moores-law-turns-55-is-it-still-relevant.

2 «Moore's Law Is Dead: Three Predictions About the Computers of Tomorrow». Был доступен: 15 сентября 2023 г. https://www.techrepublic.com/article/moores-law-is-dead-three-predictions-about-the-computers-of-tomorrow.

3 Джули Кловер (Juli Clover), «Apple's M1 Pro Chip: Everything You Need to Know», 3 ноября 2021 г. Был доступен: 15 сентября 2023 г. https://www.macrumors.com/guide/m1-pro.

4 «Apple M1». Был доступен: 15 сентября 2023 г. https://ru.wikipedia.org/wiki/Apple_M1.

5 «Intel® Xeon Phi™ Processors». Был доступен: 15 сентября 2023 г. https://ark.intel.com/content/www/us/en/ark/products/series/132784/intel-xeon-phi-72x5-processor-family.html.

6 Антон Шилов (Anton Shilov), «Intel's Sapphire Rapids Could Have 72 - 80 Cores, According to New Die Shots», 30 апреля 2021 г. Был доступен: 15 сентября 2023 г. https://www.tomshardware.com/news/intel-sapphire-rapids-could-feature-80-cores.

7 Джоэл Груска (Joel Hruska), «Future 256-Core AMD Epyc CPU Might Sport Remarkably Low 600W TDP», 1 ноября 2021 г. Был доступен: 15 сентября 2023 г. https://www.extremetech.com/computing/328692-future-256-core-amd-epyc-cpu-might-sport-remarkably-low-600w-tdp.

По мере распространения многоядерных систем интерес к многопоточности быстро растет. Стандартные средства многопоточности в языке C++ стали одним из важнейших обновлений, появившихся в версии C++11. Каждый очередной стандарт C++ добавлял возможности более высокого уровня, упрощающие разработку многопоточных приложений. Глава 17 рассказывает о том, как писать на C++ многопоточные приложения. В главе 18 представлены корутины (C++20), позволяющие писать многопоточные приложения в простом стиле последовательного кодирования.

1.4. Что такое объектно-ориентированное программирование?

Быстро, безошибочно и экономично писать программы — это очень трудная задача, тем более что сейчас быстро растет спрос на новое, все более мощное программное обеспечение. Программы состоят из многократно используемых компонентов, которые называются **объектами (objects)**, а точнее говоря (как мы покажем в главе 9), **классами (classes)**, из которых программист создает эти объекты. Вот типичные примеры: объект-автомобиль, объект-человек, объект-изображение, объект-звук, объект-дата, объект-время и т. д. Почти любое существительное («автомобиль», «человек», «изображение»...) можно логично представить в виде объекта, имеющего **атрибуты (attributes)** и **поведение (behaviors)**. Примеры атрибутов: название, размер, цвет и т. д. Примеры поведения: перемещение, общение, вычисление и т. д. Программисты выяснили, что модульный, объектно-ориентированный подход к проектированию и внедрению программного обеспечения делает их рабочие группы гораздо более продуктивными, чем при работе прежними методами. Дело в том, что объектно-ориентированные программы, как правило, легче понимать, исправлять и модифицировать.

Автомобиль как объект

Начнем с простой аналогии. Предположим, вы управляете автомобилем и хотите заставить его двигаться быстрее, нажав на педаль газа. Что должно произойти, прежде чем вы сможете это сделать? Сначала кто-то должен спроектировать этот автомобиль. Его проектируют на инженерных чертежах, напоминающих проект планировки квартиры. Чертежи подробно описывают, как устроена педаль газа. Она скрывает от водителя сложные механизмы, заставляющие автомобиль двигаться быстрее. Педаль тормоза скрывает механизмы, замедляющие ход автомобиля. Руль скрывает механизмы, поворачивающие колеса автомобиля. Все это позволяет людям, практически ничего не знающим о работе двигателя, тормозных и рулевых механизмов, легко управлять автомобилем.

Когда автомобиль спроектирован, его надо изготовить по инженерным чертежам. В готовом автомобиле есть педаль газа, ускоряющая движение. Но автомобиль, к счастью, не разгоняется сам по себе, поэтому в дополнение к педали нужен еще и водитель, который будет нажимать на нее в нужный момент.

Функции и классы

Воспользуемся этим примером с автомобилем, чтобы представить вам несколько ключевых концепций объектно-ориентированного программирования. Для выполнения любой задачи в программе должна быть соответствующая функция. Функция содержит все инструкции, выполняющие эту задачу. Они скрыты от использующего функцию программиста — точно так же, как педаль газа скрывает от водителя механизмы, заставляющие автомобиль ехать быстрее. В C++ мы обычно создаем фрагмент программы, который называется **классом (class)**, и размещаем в нем весь набор функций, выполняющих задачи этого класса, — они называются **функциями класса**, или **функциями-элементами**, или **функциями-членами (member functions)**. Например, класс, представляющий банковский счет, может содержать: 1) функцию для внесения денег на счет; 2) функцию для снятия денег со счета; 3) функцию для запроса баланса счета. Класс подобен набору инженерных чертежей автомобиля, описывающих устройство его «функций»: педалей, руля и других узлов.

Создание экземпляра класса (инстанцирование)

Кто-то должен создать по чертежам хотя бы один экземпляр автомобиля, прежде чем вы сможете управлять им. Аналогично вы должны создать объект класса, чтобы ваша программа могла использовать функции этого класса для выполнения задач. Это называется **созданием экземпляра**, или **инстанцированием (instantiation)** класса, то есть созданный объект представляет собой **экземпляр (instance)** своего класса.

Повторное использование

Один и тот же набор чертежей можно повторно использовать для создания большого количества автомобилей. И точно так же вы можете **повторно использовать (reuse)** один и тот же класс для создания большого количества объектов. Повторное использование существующих классов для создания новых классов и целых программ экономит время и упрощает работу, а цифровые системы в результате становятся надежнее и эффективнее, поскольку многие существующие классы и компоненты тщательно протестированы, отлажены и оптимизированы по скорости работы. Подобно взаимозаменяемым деталям машин, которые имели решающее значение для промышленной революции, многократно используемые классы имели решающее значение для революции программного обеспечения, вызванной объектной технологией.

Сообщения и вызовы функций

Нажимая на педаль газа, вы отправляете сообщение автомобилю о том, что надо выполнить задачу: «двигаться быстрее». А при программировании вы отправляете сообщения объектам. Каждое сообщение реализуется как **вызов функции класса (member-function call)**. Такой вызов означает, что функция класса должна выполнить свою задачу. Например, банковская программа может обратиться к объекту, представляющему счет конкретного лица, чтобы вызвать функцию внесения денег на счет, увеличив его баланс на определенную сумму.

Атрибуты и элементы данных

Кроме функций, выполняющих задачи, у автомобиля есть атрибуты: цвет, количество дверей, количество бензина в баке, текущая скорость, пробег (показания одометра) и т. д. Не только функции для выполнения задач, но и атрибуты тоже заданы в чертежах (в том числе начальные нулевые значения для одометра и датчика расхода топлива). Когда вы создаете автомобиль, эти атрибуты «появляются» вместе с ним. Каждый автомобиль «видит» только свои собственные атрибуты. Например, он «видит», сколько бензина в его бензобаке, но «не видит», сколько бензина в баках других автомобилей.

Объект тоже имеет свои атрибуты и сохраняет их, пока используется в программе. Эти атрибуты заранее заданы в классе объекта (напомним, что объект — это экземпляр класса). Например, объект конкретного банковского счета имеет атрибут баланса, представляющий количество денег на этом счете. Каждый такой объект «видит» баланс на том счете, который он представляет, но «не видит» баланс других счетов. Атрибуты хранятся в **элементах данных (data members)** класса.

Инкапсуляция

Классы скрывают от доступа извне, или **инкапсулируют (encapsulate)**, атрибуты и функции объектов, созданных из этих классов. Атрибуты и функции каждого объекта тесно связаны. Объекты могут взаимодействовать друг с другом, но обычно объекту не разрешается «видеть» внутреннее устройство других объектов. Все такие детали скрыты внутри объектов. Это **сокрытие данных (information hiding)**, как мы скоро покажем, необходимо для разработки надежных, качественных программ.

Наследование

Новый класс объектов можно быстро и удобно создать путем **наследования (inheritance)**. В новом классе при этом сохраняются все свойства существующего (родительского) класса, но они могут быть доработаны программистом под конкретные задачи. Кроме того, в новый класс можно добавить и другие свойства, уникальные для этого класса. В нашей автомобильной аналогии объект класса «кабриолет» является также и объектом более общего класса «автомобиль», но со своим уникальным свойством: крыша поднимается и опускается.

Объектно-ориентированный анализ и проектирование

Скоро вы начнете программировать на C++. Как вы будете создавать **исходный код (source code)** своей программы? Возможно, как и многие программисты, вы просто включите свой компьютер и начнете печатать. Но этот подход годится только для небольших программ (например, тех, которые вы найдете в первых главах книги). А если вас попросят создать систему управления тысячами банкоматов для крупного банка? Или если вам предложат поработать в команде из нескольких тысяч разработчиков, создающих новое поколение системы управления авиаперевозками в США? В таких больших и сложных проектах нельзя сразу садиться за компьютер и писать первые строки кода.

Чтобы найти наилучшие решения, вы должны сначала детально проанализировать предстоящий проект, сформулировать требования для него (определить, что именно должна делать система) и затем спроектировать архитектуру, удовлетворяющую этим требованиям (определить, как именно система должна это делать). В идеале вы должны пройти через этот процесс и тщательно проверить всю вашу будущую архитектуру (и обратиться к другим специалистам по программному обеспечению, чтобы они тоже ее проверили) еще до того, как вы приступите к написанию первых строк кода. Если вы выполняете анализ и проектирование системы с объектно-ориентированной точки зрения, это называется **объектно-ориентированным анализом и проектированием (object-oriented analysis and design)**, или сокращенно ООАП. C++ относится к объектно-ориентированным языкам. Программирование на таком языке называется **объектно-ориентированным программированием (object-oriented programming)**, или сокращенно ООП, и позволяет реализовать объектно-ориентированную архитектуру в виде работающей системы.

1.5. Итоги

В этой главе вы узнали, как компилировать и запускать приложения, используя три компилятора: Visual C++ из Microsoft Visual Studio (в Windows), Clang C++ из Xcode (в macOS) и g++ из GNU Compiler Collection (в Linux). Мы рассказали о ресурсе Microsoft для установки Ubuntu Linux в подсистеме Windows для Linux, позволяющей запускать g++ в Windows. Мы также продемонстрировали работу с кросс-платформенными контейнерами Docker, позволяющими запускать новейшие версии g++ и clang++ в Windows, macOS и Linux.

Мы назвали несколько ресурсов для изучения истории и архитектуры C++, в том числе ресурсы, предоставленные Бьерном Страуструпом, создателем C++. Мы обсудили закон Мура, многоядерные процессоры и важную роль конкурентного программирования для использования их преимуществ. Наконец, мы предоставили краткий обзор концепций и терминов объектно-ориентированного программирования, которые будут встречаться вам на протяжении всей книги.

В следующей главе мы покажем, как программировать на C++, используя базовые инструкции ввода и вывода, фундаментальные типы данных, арифметические операции и инструкции принятия решений. Кроме того, мы продемонстрируем удобство применения готовых объектов на примере класса `string` стандартной библиотеки C++.



Азы программирования на C++

В этой главе вы:

- напишете простейшие приложения на C++;
- научитесь использовать инструкции ввода и вывода;
- познакомитесь с основными типами данных;
- научитесь использовать арифметические операторы;
- узнаете о приоритетах арифметических операторов;
- научитесь использовать инструкции принятия решений;
- научитесь использовать операторы равенства и сравнения;
- начнете работать с объектами класса `string` стандартной библиотеки C++.

2.1. Введение

В этой главе представлено несколько примеров исходного кода, которые показывают, как выводить на экран текст и получать от пользователя данные для последующей обработки. В первых трех примерах наши программы выводят текстовые сообщения. Четвертый пример знакомит вас с арифметическими операторами C++: программа получает два числа, введенных с клавиатуры, вычисляет их сумму и выводит результат. Пятый пример демонстрирует процесс принятия решений: программа сравнивает два числа, а затем выводит сообщения в зависимости от результатов сравнения.

Учебный метод готовых объектов

В своих программах вы будете создавать и использовать множество объектов из общедоступных классов, которые тщательно разработаны и протестированы. Они позволят вам выполнять сложные задачи в минимальном объеме кода. Эти классы находятся:

- в стандартной библиотеке C++;
- в библиотеках для определенных платформ (например, в библиотеке Microsoft, предназначенной для разработки приложений Windows, или в библиотеке Apple, предназначенной для разработки приложений macOS);
- в бесплатных сторонних библиотеках с открытым исходным кодом, созданных сообществами программистов (такие сообщества поддержки есть у всех основных современных языков программирования).

Мы поможем вам быстро оценить все преимущества этого метода программирования. Перед тем как вы напишете ваши первые классы самостоятельно, вы сможете легко создавать и использовать объекты из классов стандартной библиотеки C++. Мы называем это методом готовых объектов (the Objects Natural approach). В конце этой главы вы создадите и проверите в работе несколько объектов класса `string`. Во второй половине книги вы приступите к созданию своих собственных классов и увидите, что C++ позволяет создавать классы и для ваших индивидуальных целей, и для широкого использования в сообществах программистов.

Компиляция и запуск программ

В главе 1 (в разделах 1.2.1–1.2.5) приведены инструкции по компиляции и запуску программ с помощью Microsoft Visual Studio, Apple Xcode и GNU g++. Кроме того, мы приготовили для вас видеоинструкции на сайте издательства Deitel:

http://deitel.com/c-plus-plus-20-for-programmers

2.2. Первая программа на C++: вывод текста на экран

Рассмотрим простую программу, выводящую на экран текстовое сообщение (илл. 2.1). Номера строк не являются частью программы.

```
 1 // fig02_01.cpp
 2 // Программа, выводящая сообщение
 3 #include <iostream> // Этот заголовок нужен для вывода данных на экран
 4
 5 // Выполнение программы начинается с функции main
 6 int main() {
 7    std::cout << "Welcome to C++!\n"; // Выводим сообщение
 8
 9    return 0; // Указываем, что программа завершилась успешно
10 } // Конец функции main
```

```
Welcome to C++!
```

Илл. 2.1 | Программа, выводящая сообщение

Комментарии

Строки 1 и 2:

```
// fig02_01.cpp
// Программа, выводящая сообщение
```

начинаются с двух символов слеша //. Они означают, что вся последующая часть строки — это игнорируемый компилятором **комментарий (comment)**. В комментарии первой строки указано имя файла, как и во всех остальных наших программах. Комментарий во второй строке описывает назначение программы. Комментарии, начинающиеся с символов //, называются **однострочными (single-line)** и заканчиваются в конце строки. Вы можете добавлять в исходный код не только однострочные, но и **многострочные (multiline)** комментарии, помещая их между символами /* и */, как в следующих примерах:

```
/* fig02_01.cpp: программа, выводящая сообщение */
```

или:

```
/* fig02_01.cpp
Программа, выводящая сообщение */
```

Директива препроцессора #include

Строка 3:

```
#include <iostream> // Этот заголовок нужен для вывода данных на экран
```

это **директива препроцессора (preprocessing directive)**, то есть сообщение препроцессору C++, который вызывается компилятором перед началом компиляции программы. Директива требует от препроцессора включить в программу содержимое **заголовочного файла (header file)**, или сокращенно **заголовка (header)** **<iostream>**. Этот файл содержит информацию, необходимую для компиляции любой программы, которая выводит данные на экран или получает данные с клавиатуры, используя потоки **ввода-вывода (input/output)** C++. Программа, показанная на илл. 2.1, выводит на экран текстовые данные. Заголовочные файлы детально обсуждаются в главе 5.

Пустые строки и пробелы

Строка 4 — это просто пустая строка. Для удобства чтения исходного кода вы можете вставлять в него пустые строки, пробелы и символы табуляции. Их общее название — **символы пропуска (whitespaces)**, они игнорируются компилятором.

Функция main

Строка 6:

```
int main() {
```

это неотъемлемая часть каждой программы на C++. Круглые скобки после `main` указывают на то, что это **функция (function)**. Программа на C++ обычно состоит из многих функций и классов. Но в каждой программе одна «главная» функция должна называться `main` — именно с нее начинается выполнение программы на C++. **Ключевое слово (keyword)** `int` указывает, что после своего завершения функция возвращает целочисленное значение (целое число). Ключевые слова — это лексические единицы языка C++, у каждого из них есть свое назначение. В главе 3 приведен полный список ключевых слов C++. В главе 5 мы объясним, что значит «возвращать значение», когда покажем вам, как создавать ваши собственные функции. А пока просто добавляйте ключевое слово `int` слева от имени `main` в каждой из ваших программ.

Левая фигурная скобка ({ в конце строки 6) должна начинать **тело (body)** каждой функции, то есть последовательность выполняемых функцией инструкций. Соответствующая правая фигурная скобка (} в строке 10) должна заканчивать тело каждой функции.

Инструкция вывода

Строка 7:

```
std::cout << "Welcome to C++!\n"; // Выводим сообщение
```

выводит на экран все символы, которые находятся между открывающими и закрывающими кавычками. Кавычки и символы между ними называются **строкой (string)**, или **строкой символов (character string)**, или **строковым литералом (string literal)**. Как правило, мы будем называть символы, заключенные в двойные кавычки, просто строками[1]. Символы пропуска в строках не игнорируются компилятором.

В нашей программе вся строка 7 (кроме комментария), включая `std::cout`, **оператор (operator)** `<<`, строку `"Welcome to C++!\n"` и завершающую точку

[1] Словом «string» в английском языке называют строку символов, а словом «line» — строку исходного кода программы. Оба слова переводятся на русский язык одинаково: «строка», поэтому для правильного понимания важен контекст. В этой книге строки кода пронумерованы, а строки символов заключены в кавычки или обозначены именами, поэтому различия между ними очевидны. — *Примеч. ред.*

с запятой (`;`), — это **инструкция (statement)**[1]. Почти все инструкции C++ заканчиваются точкой с запятой. Отсутствие точки с запятой в конце инструкции, когда она необходима, — это синтаксическая ошибка. Директивы препроцессора (такие, как `#include`) не являются инструкциями C++ и не заканчиваются точкой с запятой.

Обычно вывод и ввод в C++ реализуются с помощью **потоков (streams)**. Когда в строке 7 выполняется инструкция, она отправляет всю последовательность символов `"Welcome to C++!\n"` (кроме кавычек) **стандартному объекту потока вывода (standard output stream object)** `std::cout`, который «подключен» к экрану.

Отступы для наглядности кода

Когда вы пишете код программы, делайте отступ в теле каждой функции на одном уровне с фигурными скобками, ограничивающими тело. Это структурирует код и облегчает его понимание. Выберите на свое усмотрение длину отступа (количество последовательных пробелов) и соблюдайте эту длину во всей программе. Для отступов можно использовать и клавишу табуляции, но не забывайте, что длина шагов табуляции может различаться. Мы предпочитаем три пробела на каждый шаг табуляции.

Пространство имен std

Обозначение `std::` перед именем `cout` указывает компилятору, что это имя находится в заголовке стандартной библиотеки C++ (в данном случае — `<iostream>`). Иными словами, запись `std::cout` означает, что имя `cout` находится в **пространстве имен (namespace)** `std`[2]. Имена объектов `cin` (стандартный поток ввода) и `cerr` (стандартный поток ошибок) тоже находятся в пространстве имен `std`. Мы будем обсуждать пространства имен в главе 16. А пока просто не забывайте добавлять `std::` перед именами `cout`, `cin` и `cerr`. Скоро мы покажем вам еще два способа указывать пространство имен: с помощью объявлений `using` и директивы `using`.

Оператор вывода в поток и escape-последовательности

В инструкции `cout`[3] оператор `<<` используется в качестве **оператора вывода в поток (stream insertion operator)**. При выполнении программы значение справа от оператора, то есть правый **операнд (operand)**, отправляется в поток вывода.

---

[1] Не следует путать термины «statement» («инструкция») и «operator» («оператор»). В этом примере оператор `<<` входит в состав инструкции. Список операторов языка C++ приведен в конце книги, в приложении А. В некоторых языках программирования инструкции также называют командами. — *Примеч. ред.*

[2] Это сокращение от слова «standard».

[3] Инструкцией языка C++ является не отдельное слово «cout», а вся строка 7 (кроме комментария). И в русском, и в английском языке такие словосочетания, как «инструкция cout» («cout statement»), «инструкция if» («if statement») и т. п., являются общепринятыми сокращениями, означающими: «инструкция с ключевым словом cout», «инструкция с ключевым словом if» и т. п. — *Примеч. ред.*

Обратите внимание, что оператор << «указывает» в ту сторону, куда отправляются данные. Символы в строках обычно отображаются так же, как они написаны между двойными кавычками. Но в выведенном на экран сообщении (илл. 2.1) нет символов \n. Обратный слеш (\) называется **экранирующим символом**, или **escape-символом (escape character)**, и служит для вывода специальных символов. Когда в строке встречается обратный слеш, он объединяется со следующим символом, образуя вместе с ним **экранированную последовательность**, или **escape-последовательность (escape sequence)**. Escape-последовательность \n означает **перевод строки (newline)**, то есть перемещает курсор (индикатор текущего положения на экране) в начало следующей строки. Некоторые важные escape-последовательности показаны в таблице.

Escape-последовательность	Описание
`\n`	Перевод строки (newline). Перемещает экранный курсор в начало следующей строки
`\t`	Горизонтальная табуляция (horizontal tab). Перемещает экранный курсор в следующую позицию табуляции
`\r`	Возврат каретки (carriage return). Перемещает экранный курсор в начало текущей строки (без перехода к следующей строке)
`\a`	Предостережение (alert). Воспроизводит системный звук
`\\`	Обратный слеш (backslash). Включает в строку символ обратного слеша
`\'`	Одиночная кавычка (single quote). Включает в строку символ одинарной кавычки
`\"`	Двойная кавычка (double quote). Включает в строку символ двойной кавычки

Инструкция return

Строка 9:

```
return 0; // Указываем, что программа завершилась успешно
```

это один из нескольких способов, которые мы будем использовать для **выхода из функции (exit a function)**. Если в конце функции `main` есть инструкция `return` со значением `0`, это указывает на успешное завершение программы. Впрочем, если в процессе выполнения функция `main` достигает закрывающей фигурной скобки без инструкции `return`, компилятор рассматривает это как возврат `0` и предполагает, что программа завершилась успешно. В последующих примерах мы будем опускать `return` в функциях `main` тех программ, которые завершаются успешно.

2.3. Модификация нашей первой программы на C++

Следующие два примера модифицируют программу, показанную на илл. 2.1. Первый пример выводит текст одной строкой с помощью нескольких инструкций. Второй — выводит текст в несколько строк с помощью одной инструкции.

Вывод одной строки текста несколькими инструкциями

На илл. 2.2 показан пример вывода в поток двумя последовательными инструкциями (строки 7–8). При этом на экране отобразится то же самое, что на илл. 2.1. Каждый раз, когда вы выводите данные в поток, символы «печатаются» на экране с того места, где завершился предыдущий вывод. В строке 7 выводится слово `Welcome` с последующим пробелом. После пробела нет escape-последовательности `\n` (перевод строки), поэтому инструкция в строке 8 сразу же «допечатает» окончание фразы.

```
1 // fig02_02.cpp
2 // Вывод одной строки текста несколькими инструкциями
3 #include <iostream> // Этот заголовок нужен для вывода данных на экран
4
5 // Выполнение программы начинается с функции main
6 int main() {
7    std::cout << "Welcome ";
8    std::cout << "to C++!\n";
9 } // Конец функции main
```

```
Welcome to C++!
```

Илл. 2.2 | Вывод одной строки текста несколькими инструкциями

Вывод нескольких строк текста одной инструкцией

Одна инструкция может выводить текст в несколько строк («в столбик»), если включить в нее дополнительные escape-последовательности `\n`, как в строке 7 на илл. 2.3. Каждый раз, когда в потоке вывода встречается `\n`, экранный курсор перемещается в начало следующей строки. Чтобы получить пустую строку, выведите `\n` два раза подряд.

```
1 // fig02_03.cpp
2 // Вывод нескольких строк текста одной инструкцией
3 #include <iostream> // Этот заголовок нужен для вывода данных на экран
4
5 // Выполнение программы начинается с функции main
6 int main() {
7    std::cout << "Welcome\nto\n\nC++!\n";
8 } // Конец функции main
```

```
Welcome
to

C++!
```

Илл. 2.3 | Вывод нескольких строк текста одной инструкцией

2.4. Вторая программа на C++: сложение целых чисел

Наша следующая программа получает два целых числа с клавиатуры, вычисляет их сумму и выводит результат на экран через `std::cout`. На илл. 2.4 вы видите исходный код программы и пример ввода и вывода. Вводимые пользователем данные выделены **жирным шрифтом**.

```
1 // fig02_04.cpp
2 // Программа, выводящая сумму двух целых чисел
3 #include <iostream> // Этот заголовок нужен для ввода и вывода данных
4
5 // Выполнение программы начинается с функции main
6 int main() {
7    // Объявляем и инициализируем переменные
8    int number1{0}; // Первая переменная типа int (начальное значение: 0)
9    int number2{0}; // Вторая переменная типа int (начальное значение: 0)
10   int sum{0}; // Сумма number1 и number2 (начальное значение: 0)
11
12   std::cout << "Enter first integer: "; // Ждем ввода данных
13   std::cin >> number1; // Считываем первое число и помещаем его в number1
14
15   std::cout << "Enter second integer: "; // Ждем ввода данных
16   std::cin >> number2; // Считываем второе число и помещаем его в number2
17
18   sum = number1 + number2; // Суммируем числа и записываем результат в sum
19
20   std::cout << "Sum is " << sum << "\n"; // Выводим результат
21 } // Конец функции main
```

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

Илл. 2.4 | Программа, выводящая сумму двух целых чисел

Объявления переменных с инициализацией

Строки 8–10:

```
int number1{0}; // Первая переменная типа int (начальное значение: 0)
int number2{0}; // Вторая переменная типа int (начальное значение: 0)
int sum{0}; // Сумма number1 и number2 (начальное значение: 0)
```

содержат **объявления (declarations)** трех **переменных (variables)** с именами `number1`, `number2` и `sum`. Каждая из этих переменных объявляется с указанием того, что она имеет **тип данных (data type)** `int` (имя этого типа является сокращением от слова «integer» — «целое число»). В переменных типа `int` можно хранить только целые числа, например `7`, `-11`, `0` или `31914`. В языке C++ каждую переменную обязательно надо объявлять, указывая ее имя и тип данных.

В строках 8–10 переменные не только объявляются, но и инициализируются значением `0` (то есть каждой из них присваивается значение `0`). Обратите внимание, что значение должно быть указано в фигурных скобках { } сразу после имени переменной. Эта форма записи называется **скобочной инициализацией (braced initialization)**, она была введена в C++11. Хотя не всегда необходимо явно инициализировать каждую переменную, запомните, что инициализация поможет вам избежать многих проблем!

До C++11 строки 8–10 выглядели бы так:

```
int number1 = 0; // Первая переменная типа int (начальное значение: 0)
int number2 = 0; // Вторая переменная типа int (начальное значение: 0)
int sum = 0; // Сумма number1 и number2 (начальное значение: 0)
```

С этим традиционным способом инициализации вы встретитесь, если будете изучать исходные коды старых программ на C++. У скобочной инициализации есть ряд преимуществ, которые мы обсудим в последующих главах.

Одновременное объявление нескольких переменных

Несколько переменных одного типа разрешается объявлять в одной инструкции. Например, вы могли бы объявить и инициализировать все три переменные следующим образом:

```
int number1{0}, number2{0}, sum{0};
```

Но эта форма записи обычно делает исходный код менее наглядным и затрудняет добавление комментариев, поясняющих смысл каждой переменной.

Основные типы данных

Скоро мы рассмотрим тип `double`, предназначенный для вещественных чисел, и тип `char`, предназначенный для символьных данных. Вещественные числа — это числа с десятичной точкой[1], например `3.4`, `0.0` или `-11.19`. Переменная типа `char` может содержать только одну строчную букву, или прописную букву, или цифру, или специальный символ (например, `$` или `*`). Типы данных `int`, `double`, `char` и `long long` относятся к **основным**, или **фундаментальным**, **типам (fundamental types)** языка C++. Имена основных типов данных состоят из одного или нескольких ключевых слов и пишутся строчными буквами. Здесь вы найдете полный список основных типов данных языка C++ и их допустимых диапазонов:

https://en.cppreference.com/w/cpp/language/types

Идентификаторы и имена в стиле «верблюжий регистр»

Вы можете давать переменным любые имена, которые являются допустимыми идентификаторами (например, `number1`). **Идентификатор (identifier)** — это после-

[1] В русском издании книги сохранен американский стандарт записи чисел, в котором десятичным разделителем является точка, а не запятая. — *Примеч. ред.*

довательность символов, состоящая из букв, цифр и символов подчеркивания (_), которая не начинается с цифры и не является ключевым словом. Синтаксис языка C++ **чувствителен к регистру (case sensitive)**: прописные и строчные буквы различаются, то есть `a1` и `A1` — это разные идентификаторы.

C++ поддерживает идентификаторы любой длины. Не начинайте идентификатор с символа подчеркивания и последующей заглавной буквы (`_A`, `_B`, `_C`...), или с двух символов подчеркивания (`__`). Компиляторы C++ используют такие имена для собственных нужд.

По общепринятому соглашению, имя каждой переменной должно начинаться со строчной буквы, а каждое последующее слово в этом имени должно начинаться с заглавной буквы. Например, вы можете назвать переменную именем `firstNumber`, в котором второе слово `Number` пишется с заглавной буквы `N`. Это соглашение об именах известно как **«верблюжий регистр» (camel case)**, потому что заглавные буквы выделяются подобно горбам верблюда: `camelCase`.

Размещение объявлений переменных

Переменную можно объявить практически в любом месте программы, но обязательно до того, как вы впервые обратитесь к этой переменной (перед тем, как вы начнете ее использовать). Например, строку 8:

```
int number1{0}; // Первая переменная типа int (начальное значение: 0)
```

можно было бы разместить непосредственно перед строкой 13:

```
std::cin >> number1; // Считываем первое число и помещаем его в number1
```

Объявление переменной в строке 9:

```
int number2{0}; // Вторая переменная типа int (начальное значение: 0)
```

можно было бы разместить перед строкой 16:

```
std::cin >> number2; // Считываем второе число и помещаем его в number2
```

А строку 10:

```
int sum{0}; // Сумма number1 и number2 (начальное значение: 0)
```

можно было бы разместить непосредственно перед строкой 18:

```
sum = number1 + number2; // Суммируем числа и записываем результат в sum
```

Или вы могли бы объединить строки 10 и 18 в следующее объявление, размещенное непосредственно перед строкой 20:

```
int sum{number1 + number2}; // Инициализируем sum суммой number1 + number2
```

Получение первого значения от пользователя

Строка 12:

```
std::cout << "Enter first integer: "; // Ждем ввода данных
```

выводит последовательность символов `Enter first integer:` с последующим пробелом. Такое сообщение называется **приглашением (prompt)**, поскольку оно показывает пользователю, что программа ждет от него нужного действия.

Строка 13:

```
std::cin >> number1; // Считываем первое число и помещаем его в number1
```

использует **стандартный объект потока ввода (standard input stream object)** `cin` (из пространства имен `std`) и **оператор извлечения из потока (stream extraction operator)** `>>` для ввода данных с клавиатуры.

После выполнения предыдущей инструкции программа ожидает ввода значения переменной `number1`. Пользователь вводит целое число (в виде символов), затем нажимает клавишу `Enter` (иногда она называется `Return`), чтобы отправить эти символы в программу. Объект `cin` преобразует символьное представление числа в целочисленное значение (`int`) и присваивает это значение переменной `number1`. Нажатие клавиши `Enter` также приводит к перемещению курсора в начало следующей строки на экране.

Когда программа ожидает ввода целого числа, пользователь может набрать на клавиатуре что-то другое: буквы, случайные символы (например, `#` или `@`) или число с десятичной точкой (например, `73.5`). Но в первых учебных программах мы исходим из того, что пользователь вводит корректные данные. Позже мы расскажем, как решать проблемы, связанные с вводом некорректных данных.

Получение второго значения от пользователя

Строка 15:

```
std::cout << "Enter second integer: "; // Ждем ввода данных
```

выводит последовательность символов `Enter second integer:` с последующим пробелом и ждет реакции пользователя.

Строка 16:

```
std::cin >> number2; // Считываем второе число и помещаем его в number2
```

получает от пользователя значение для переменной `number2`.

Вычисление суммы чисел, введенных пользователем

Выражение присваивания в строке 18:

```
sum = number1 + number2; // Суммируем числа и записываем результат в sum
```

суммирует значения переменных `number1` и `number2` и присваивает результат переменной `sum` с помощью **оператора присваивания (assignment operator)** `=`. Большинство вычислений выполняется в выражениях присваивания. Операторы `=` и `+` являются **бинарными (binary operators)**, поскольку каждый из них имеет два операнда. Для оператора `+` операндами являются `number1` и `number2`. Для оператора `=` операндами являются переменная `sum` и значение выражения `number1 + number2`.

Добавление пробелов до и после бинарного оператора выделяет его визуально, поэтому исходный код становится проще для чтения.

Вывод результата на экран

Строка 20:

```
std::cout << "Sum is " << sum << "\n"; // Выводим результат
```

выводит строку `"Sum is"`, за которой следуют пробел, значение переменной `sum` и перевод строки.

Эта инструкция выводит несколько значений, имеющих разные типы. Оператор вывода в поток «знает», как надо выводить данные каждого типа. Инструкция с несколькими последовательными операторами вывода в поток (`<<`) выполняет **конкатенацию (concatenating)**, или **каскадирование (cascading)**, операций вывода.

Вычисления могут выполняться непосредственно в инструкциях вывода в поток. Мы могли бы исключить переменную `sum`, объединив строки 18 и 20:

```
std::cout << "Sum is " << number1 + number2 << "\n";
```

Очень важно, что C++ открывает перед вами возможность создания ваших собственных типов данных, называемых классами (мы обсудим их в начале главы 9). Вы можете «научить» C++ вводу и выводу значений ваших новых типов, используя операторы `>>` и `<<` соответственно. Это называется **перегрузкой операторов (operator overloading)**, мы рассмотрим ее в главе 11.

2.5. Арифметические операторы

В следующей таблице приведены **арифметические операторы (arithmetic operators)** C++:

Операция	Арифметический оператор	Примеры алгебраических выражений	Примеры выражений C++
Сложение	`+`	$f + 7$	`f + 7`
Вычитание	`-`	$p - c$	`p - c`
Умножение	`*`	bm $b \cdot m$	`b * m`
Деление	`/`	x/y $x \div y$	`x / y`
Остаток от деления	`%`	$r \bmod s$	`r % s`

Обратите внимание на несколько символов, которых нет в алгебре. Звездочка (`*`) означает умножение, а знак процента (`%`) — это оператор остатка от деления, который мы скоро рассмотрим. Все эти арифметические операторы являются бинарными.

Скачать БЛАГОДАРЯ BoomInfo
BOOMINFO.PRO
Открой сокровищницу ИНФО пиратов

Целочисленное деление

Целочисленное деление (integer division), в котором числитель и знаменатель являются целыми числами, дает в результате целочисленное частное. Например, выражение `7 / 4` вычисляется как `1`, а выражение `17 / 5` вычисляется как `3`. Любая дробная часть, оставшаяся в результате целочисленного деления, просто отбрасывается, при этом не происходит округления.

Оператор остатка от деления

Оператор `%` — это **оператор остатка от деления (remainder operator)**, также называемый **оператором модуля (modulus operator)**. Он вычисляет остаток после целочисленного деления и может работать только с целочисленными операндами, то есть результатом выражения `x % y` будет остаток после деления `x` на `y`. Таким образом, результат выражения `7 % 4` равен `3`, а результат выражения `17 % 5` равен `2`.

Круглые скобки для группировки выражений

Круглые скобки в выражениях C++ используются так же, как в алгебре. Например, чтобы умножить `a` на величину `b + c`, мы напишем: `a * (b + c)`.

Правила приоритета операторов

C++ определяет приоритеты операторов в арифметических выражениях по таким же **правилам приоритета (rules of precedence)**, как в алгебре:

1. Выражения в круглых скобках вычисляются первыми. Круглые скобки имеют высший приоритет. В случаях вложенных круглых скобок, например

   ```
   (a * (b + c))
   ```

   выражения в самой внутренней паре круглых скобок вычисляются первыми.

2. Затем выполняются операции умножения, деления и остатка от деления. В выражениях, содержащих несколько подобных операций, эти вычисления выполняются слева направо. Операторы умножения, деления и остатка от деления имеют одинаковый приоритет.

3. Операции сложения и вычитания выполняются в последнюю очередь. Если выражение содержит несколько таких операций, то они выполняются слева направо. Операторы сложения и вычитания тоже имеют одинаковый приоритет.

В конце книги, в приложении A, приведена полная таблица приоритетов операторов. Обратите внимание: в таких выражениях, как `(a + b) * (c - d)`, где две пары круглых скобок не являются вложенными и имеют одинаковый приоритет, стандарт C++ не определяет порядок вычисления выражений в круглых скобках!

Группировка операторов

Когда мы говорим, что в языке C++ определенные операторы применяются слева направо, мы имеем в виду **группировку (grouping)** операторов, которую также называют **ассоциативностью (associativity)** операторов. Например, в выражении:

```
a + b + c
```

BOOMINFO.PRO
Открой сокровищницу ИНФО пиратов

операторы сложения (+) группируются слева направо, как если бы мы добавили скобки: (a + b) + c. Большинство операторов C++, имеющих одинаковый приоритет, группируются слева направо. Позже мы расскажем о некоторых операторах, которые группируются справа налево.

2.6. Принятие решений: операторы равенства и сравнения

Теперь мы познакомим вас с инструкцией `if`, которая позволяет программе выполнять альтернативные действия в зависимости от того, является ли заданное **условие (condition)** истинным или ложным. Условия в инструкциях `if` могут быть сформированы с помощью перечисленных в таблице **операторов равенства (equality operators)** и **операторов сравнения (relational operators)**, также называемых **операторами отношения**, или **реляционными операторами**:

Оператор равенства или сравнения в алгебре	Оператор равенства или сравнения в C++	Пример условия в C++	Смысл условия
Операторы сравнения			
>	>	x > y	x больше y
<	<	x < y	x меньше y
≥	>=	x >= y	x больше или равно y
≤	<=	x <= y	x меньше или равно y
Операторы равенства			
=	==	x == y	x равно y
≠	!=	x != y	x не равно y

Все операторы сравнения имеют одинаковый приоритет и группируются слева направо. Оба оператора равенства имеют одинаковый приоритет, который ниже приоритета операторов сравнения. Операторы равенства тоже группируются слева направо.

Если перепутать местами два символа в операторах !=, >= и <= (то есть написать их как =!, => и =<), это обычно считается синтаксической ошибкой. В некоторых случаях запись =! вместо != не считается синтаксической ошибкой, но почти наверняка будет логической ошибкой, которая проявится во время выполнения программы. Вы узнаете причины этого при рассмотрении логических операторов в разделе 4.11.

Принципиальное различие между операторами == и =

Если вместо оператора равенства == вы поставите оператор присваивания =, то логическая ошибка вам гарантирована. В устной речи оператор равенства обычно

называют «равно» или «двойное равно», а оператор присваивания — «получает», «получает значение» или «присваивается значение». Если перепутать эти операторы, это не обязательно приведет к легко распознаваемой синтаксической ошибке, зато может привести к незаметным с первого взгляда логическим ошибкам. Компиляторы обычно предупреждают об этом.

Инструкция if

На илл. 2.5 показано, как шесть инструкций `if` последовательно сравнивают два целых числа, введенных пользователем. Если условие в инструкции `if` истинно, то выполняется ее тело (соответствующая инструкция, заключенная в фигурные скобки). Если условие ложно, то тело инструкции не выполняется.

```
// fig02_05.cpp
// Сравнение целых чисел с помощью инструкций if,
// операторов равенства и сравнения
#include <iostream> // Этот заголовок нужен для ввода и вывода

using std::cout; // Программа использует cout
using std::cin; // Программа использует cin

// Выполнение программы начинается с функции main
int main() {
   int number1{0}; // Первое число для сравнения (начальное значение: 0)
   int number2{0}; // Второе число для сравнения (начальное значение: 0)

   cout << "Enter two integers to compare: "; // Ждем ввода данных
   cin >> number1 >> number2; // Считываем два введенных пользователем числа

   if (number1 == number2) {
      cout << number1 << " == " << number2 << "\n";
   }

   if (number1 != number2) {
      cout << number1 << " != " << number2 << "\n";
   }

   if (number1 < number2) {
      cout << number1 << " < " << number2 << "\n";
   }

   if (number1 > number2) {
      cout << number1 << " > " << number2 << "\n";
   }

   if (number1 <= number2) {
      cout << number1 << " <= " << number2 << "\n";
   }

   if (number1 >= number2) {
      cout << number1 << " >= " << number2 << "\n";
```

Окончание ⇨

```
Enter two integers to compare: 3 7
3 != 7
3 < 7
3 <= 7
```

```
Enter two integers to compare: 22 12
22 != 12
22 > 12
22 >= 12
```

```
Enter two integers to compare: 7 7
7 == 7
7 <= 7
7 >= 7
```

Строка 15:

```
cin >> number1 >> number2; // Считываем два введенных пользователем числа
```

использует каскадирование операций извлечения из потока, чтобы получить два числа от пользователя. Напомним, что в строке 7 было объявление `using`, поэтому теперь можно писать сокращенно: `cin` (вместо `std::cin`). Сначала считывается значение в переменную `number1`, затем — в переменную `number2`.

Сравнение чисел

Инструкция `if` в строках 17–19:

```
if (number1 == number2) {
   cout << number1 << " == " << number2 << "\n";
}
```

проверяет, равны ли `number1` и `number2`. Если равны, то инструкция `cout` выводит сообщение об этом. Таким же образом в остальных инструкциях `if` (начинающихся в строках 21, 25, 29, 33 и 37) инструкция `cout` выводит соответствующее сообщение, если условие окажется истинным (`true`).

Фигурные скобки и блоки

Для наглядности кода, показанного на илл. 2.5, мы поместили в **тело (body)** каждой инструкции `if` только одну инструкцию `cout`. Обратите внимание, что каждая инструкция `cout` находится между фигурными скобками { }. Одна или несколько инструкций, расположенных между фигурными скобками, образуют **составную инструкцию (compound statement)**, или **блок (block)**. Если тело состоит только из одной инструкции, как в нашем примере, то эти скобки необязательны. Но во всех случаях, когда тело состоит из нескольких инструкций, их необходимо объединять в блок, то есть заключать в фигурные скобки. Если забыть о скобках, это приведет к ошибкам. Чтобы избежать ошибок, возьмите себе за правило: тело каждой инструкции `if` (даже самой простой) надо заключать в фигурные скобки.

Частая логическая ошибка: точка с запятой после условия

Если случайно поставить точку с запятой сразу после правой круглой скобки условия в инструкции `if`, то это будет логической ошибкой (но не будет синтаксической ошибкой). Точка с запятой сразу после условия означает, что у инструкции `if` пустое тело, то есть она не выполнит никаких действий — независимо от того, окажется ли ее условие истинным или ложным. А настоящее тело инструкции `if` превратится в отдельную инструкцию, то есть будет выполняться в любом случае. Это приведет к неправильной работе программы. Впрочем, большинство компиляторов при обнаружении этой логической ошибки выдают предупреждение.

Многострочная запись длинных инструкций

Длинную инструкцию можно записать в несколько строк. В таких случаях выбирайте осмысленные места переноса на следующую строку. Например, делайте

переносы после запятых в списке или после операторов в длинном выражении. Когда инструкцию записывают в несколько строк, принято делать одинаковые отступы от левого края во всех этих строках.

Приоритеты и группировка операторов

Кроме операторов присваивания (=), все остальные операторы, представленные в этой главе, группируются слева направо. Операторы присваивания (=) группируются справа налево. Например, выражение `x = y = 0` вычисляется так, как если бы в правой части были скобки: `x = (y = 0)`. То есть сначала переменной `y` присваивается значение `0`, а затем результат (`0`) присваивается переменной `x`.

Составляя выражение, содержащее много операторов, сверяйтесь с полной таблицей приоритетов в конце книги, в приложении А. Убедитесь, что операторы в выражении группируются именно в том порядке, который вам нужен. Если вы не знаете наверняка, в каком порядке будет вычисляться сложное выражение, разделите его на несколько инструкций или используйте круглые скобки, чтобы задать явный порядок вычислений, как это принято в алгебре.

2.7. Готовые объекты: создание и использование объектов класса string стандартной библиотеки

На страницах этой книги мы будем показывать вам, насколько полезны классы стандартной библиотеки C++ и классы многочисленных библиотек с открытым исходным кодом, созданных в сообществе C++. Вы узнаете, какие библиотеки существуют, научитесь выбирать из них необходимые для ваших приложений, будете создавать объекты из стандартных библиотечных классов и пользоваться их богатыми возможностями. Говоря о готовых объектах, мы имеем в виду, что вы сможете писать программы с профессионально разработанными объектами еще до того, как научитесь создавать свои собственные классы.

Вы уже работали с объектами C++, в частности с объектами `cin` и `cout`, которые инкапсулируют механизмы ввода и вывода соответственно. Эти объекты сконструировали для вас опытные программисты — разработчики классов, входящих в заголовок `<iostream>`. В этом разделе вы будете создавать и применять объекты класса `string` стандартной библиотеки C++[1].

Знакомство с классом string

Классы не могут выполняться сами по себе. Объектом «автомобиль» управляет объект «водитель», указывая ему, что надо делать (ехать быстрее или медленнее, повернуть налево или направо). Для этого водителю не надо знать, как устроены внутренние механизмы автомобиля. Примерно так же главная функция `main` будет «управлять» объектом класса `string`, вызывая его функции. Для этого ей не надо

[1] В последующих главах вы узнаете о других возможностях класса string. В главе 8 мы подробно обсуждаем его и демонстрируем многие его функции.

знать, как устроен класс `string`. В этом смысле функцию `main` в нашем следующем примере можно назвать управляющей программой. На илл. 2.6 вы видите, как из функции `main` вызываются несколько функций класса `string`.

```
// fig02_06.cpp
// Тестирование класса string стандартной библиотеки
#include <iostream>
#include <string>
using namespace std;

int main() {
   string s1{"happy"};
   string s2{" birthday"};
   string s3; // Создаем пустую строку

   // Отображаем строки и их длину
   cout << "s1: \"" << s1 << "\"; length: " << s1.length()
      << "\ns2: \"" << s2 << "\"; length: " << s2.length()
      << "\ns3: \"" << s3 << "\"; length: " << s3.length();

   // Сравниваем строки операторами == и !=
   cout << "\n\nThe results of comparing s2 and s1:" << boolalpha
      << "\ns2 == s1: " << (s2 == s1)
      << "\ns2 != s1: " << (s2 != s1);

   // Тестируем функцию empty класса string
   cout << "\n\nTesting s3.empty():\n";

   if (s3.empty()) {
      cout << "s3 is empty; assigning to s3;\n";
      s3 = s1 + s2; // Записываем в s3 результат конкатенации s1 и s2
      cout << "s3: \"" << s3 << "\"";
   }

   // Тестируем новые функции класса string (C++20)
   cout << "\n\ns1 starts with \"ha\": " << s1.starts_with("ha") << "\n";
   cout << "s2 starts with \"ha\": " << s2.starts_with("ha") << "\n";
   cout << "s1 ends with \"ay\": " << s1.ends_with("ay") << "\n";
   cout << "s2 ends with \"ay\": " << s2.ends_with("ay") << "\n";
}
```

```
s1: "happy"; length: 5
s2: " birthday"; length: 9
s3: ""; length: 0

The results of comparing s2 and s1:
s2 == s1: false
s2 != s1: true

Testing s3.empty():
s3 is empty; assigning to s3;
s3: "happy birthday"
```

Окончание ⇨

```
s1 starts with "ha": true
s2 starts with "ha": false
s1 ends with "ay": false
s2 ends with "ay": true
```

Илл. 2.6 | Тестирование класса string стандартной библиотеки

Создание объектов

Как правило, вы не можете использовать функции какого-либо класса, пока не создадите объект этого класса (также называемый экземпляром класса)[1]. В строках 8–10 создаются три объекта класса `string`:

- объект `s1` инициализируется копией строкового литерала `"happy"`;
- объект `s2` инициализируется копией строкового литерала `" birthday"`;
- объект `s3` по умолчанию инициализируется пустой строкой (`""`).

Когда мы объявляем переменные типа `int` (как мы уже делали), компилятор знает, что `int` — это один из основных (фундаментальных) типов, встроенных в C++. Но тип `string` в строках 8–10 компилятору заранее неизвестен: это **тип класса** (**class type**) стандартной библиотеки C++.

Правильно упакованные классы могут повторно использоваться другими программистами. Это одно из главных преимуществ таких объектно-ориентированных языков, как C++, расширяемых мощными библиотеками готовых классов. Например, вы можете повторно использовать все классы стандартной библиотеки C++ в любой своей программе, включив в нее соответствующие заголовки директивой `#include`. В нашем случае это заголовок `<string>` (строка 4). Имя `string`, как и имя `cout`, находится в пространстве имен `std`.

Функция length класса string

Строки 13–15 по очереди выводят на экран каждый из объектов `string` (содержащуюся в нем строку символов и ее длину). Имеющаяся в классе `string` функция `length` возвращает количество символов, хранящихся в конкретном объекте `string`. Например, в строке 13 выражение:

```
s1.length()
```

возвращает длину строки объекта `s1` через вызов функции `length`. Чтобы вызвать эту функцию для конкретного объекта, надо указать его имя (`s1`), затем оператор «точка» (`.`) и имя функции (`length`) с последующими круглыми скобками. Пустые круглые скобки означают, что функция `length` не требует никаких дополнительных данных для выполнения своей задачи. Вскоре вы увидите, что некоторым другим функциям классов для выполнения своих задач нужны дополнительные данные, называемые аргументами.

[1] В разделе 9.20 вы узнаете, как вызывать статические (static) функции класса, не создавая объекты этого класса.

С точки зрения главной функции `main` при вызове функции `length` происходит следующее:

1. Программа переносит выполнение из точки вызова (строка 13) в функцию `length`. Поскольку функция `length` была вызвана через объект `s1`, она «знает», что должна работать с данными именно этого объекта.
2. Функция `length` выполняет свою задачу: возвращает длину строки объекта `s1` в точку вызова (строка 13). Функция `main` не знает, как был получен этот результат. (Напомним сравнение: водитель управляет автомобилем, не зная о том, как устроены двигатель, коробка передач, рулевой механизм и тормоза.)
3. Объект `cout` выводит на экран количество символов, полученных от функции `length`, а затем программа продолжает выполнение, выводя аналогичную информацию об объектах `s2` и `s3`.

Сравнение объектов string операторами равенства

Как и числа, объекты `string` можно сравнивать друг с другом. В строках 18–20 объекты `s2` и `s1` сравниваются с помощью операторов равенства. Обратите внимание: при сравнениях объектов `string` регистр букв имеет значение[1].

Обычно, когда вы выводите на экран результат проверки какого-либо условия, это будет `0` (если условие ложно) или `1` (если оно истинно). Но обработчик потока `boolalpha` (строка 18) из заголовка `<iostream>` сообщает выходному потоку, что результат должен выводиться в виде слов `false` («ложно») или `true` («истинно»).

Функция empty класса string

В строке 25 вызывается функция `empty`, которая возвращает `true`, если объект `string` пустой (если длина строки символов равна `0`). В противном случае функция `empty` возвращает значение `false`. Объект `s3` был по умолчанию инициализирован пустой строкой, то есть в нем нет ни одного символа. Поэтому тело инструкции `if` будет выполнено.

Конкатенация и присваивание строк

В строке 27 объекту `s3` присваивается новое значение, полученное путем «сложения» строк объектов `s1` и `s2` с помощью оператора `+`. Такое «сложение» называется **конкатенацией (concatenation)** строк. После конкатенации `s3` содержит строку `"happy birthday"`, то есть все символы `s1`, а затем все символы `s2`. В строке 28 выводится на экран значение `s3` для подтверждения успешной конкатенации.

Функции starts_with и ends_with класса string (C++20)

В строках 32–35 демонстрируются две новые функции класса `string`, введенные в C++20: `starts_with` и `ends_with`. Функция `starts_with` возвращает значение

[1] В главе 8 вы узнаете, что строки сравниваются лексикографически, то есть путем сравнения числовых значений символов, из которых они состоят.

`true`, если строка объекта `string` начинается указанной последовательностью символов. Функция `ends_with` возвращает значение `true`, если эта строка заканчивается указанной последовательностью символов. В противном случае обе функции возвращают значение `false`. При выполнении строк 32 и 33 вы видите, что `s1` начинается с символов `"ha"`, в отличие от `s2`. При выполнении строк 34 и 35 вы видите, что `s1` не заканчивается символами `"ay"`, в отличие от `s2`.

2.8. Итоги

Мы рассказали о нескольких важных базовых механизмах языка C++, включая вывод данных на экран, ввод данных с клавиатуры и объявление переменных основных типов. Теперь вы знаете, как использовать объект потока ввода `cin` и объект потока вывода `cout`, необходимые в простых интерактивных программах. Вы научились объявлять и инициализировать переменные, а также выполнять вычисления с арифметическими операторами. Мы обсудили правила приоритета операторов (порядок выполнения операций) и группировку (ассоциативность) операторов. Вы узнали, как условные инструкции `if` позволяют программам принимать решения. Мы показали, как использовать в инструкциях `if` операторы равенства и сравнения, чтобы задавать проверяемые условия.

Наконец, мы представили метод изучения C++ на основе готовых объектов: вы создали несколько объектов класса `string` стандартной библиотеки C++ и поработали с ними с помощью операторов равенства и функций класса. В последующих главах вы будете создавать и использовать много разных объектов стандартных классов, чтобы выполнять важные задачи в минимальном объеме кода. В главах 9–11 вы научитесь создавать свои собственные классы и увидите, что C++ дает вам возможность получать очень ценные, уникальные «классы ручной работы». А сейчас, в главе 3, вы познакомитесь с управляющими инструкциями, от которых зависит порядок выполнения действий в программе.



Управляющие инструкции (часть 1)

В этой главе вы:

- научитесь использовать инструкции `if` и `if...else` для выбора между альтернативными действиями;
- научитесь использовать инструкцию цикла `while` для многократного выполнения других инструкций;
- научитесь использовать циклы со счетчиком и и циклы с контрольным значением;
- научитесь использовать вложенные управляющие инструкции;
- научитесь использовать составные операторы присваивания, а также операторы инкремента и декремента;
- узнаете, почему основные типы данных на разных платформах несовместимы;
- продолжите работать с готовыми объектами, оперируя с очень большими целыми числами;
- познакомитесь с синтаксисом форматирования текста в C++20: более кратким и мощным, чем в прежних версиях C++.

3.1. Введение

В этой и последующих главах мы расскажем о принципах структурного программирования. Представленные здесь концепции очень важны для создания классов и управления объектами. Мы подробно рассмотрим инструкцию `if`, а затем инструкции `if...else` и `while`. Мы также расскажем о составных операторах присваивания и об операторах инкремента и декремента.

Вы узнаете, почему фундаментальные (основные) типы данных не всегда портируются (переносятся) с одной компьютерной платформы на другую. Вы продолжите работу с готовыми объектами, на этот раз на примере операций с целыми числами произвольной величины — их значения могут выходить далеко за пределы тех диапазонов, которые компьютер способен обрабатывать на аппаратном уровне.

Мы начнем знакомить вас с появившимся в C++20 синтаксисом форматирования текста, основанным на аналогичных синтаксических средствах из языков Python, Microsoft.NET (C# и Visual Basic) и Rust[1]. По сравнению с предшествующими версиями C++ новый синтаксис стал лаконичнее и мощнее.

3.2. Управляющие структуры

В 1960-е годы стало ясно, что причина многих проблем, затрудняющих совместную работу групп программистов, — это беспорядочная передача управления от одних инструкций к другим. Виновником была объявлена инструкция `goto` (входящая тогда в большинство языков программирования), позволяющая произвольно менять последовательность выполнения других инструкций.

Исследование Бёма и Якопини[2] показало, что можно писать программы, не применяя `goto`. И программистам той эпохи пришлось постараться, чтобы изменить свой привычный стиль мышления на «программирование без `goto`». Термин **«структурное программирование» (structured programming)** стал почти синонимом всеобщего отказа от `goto`. Результаты оказались впечатляющими. Многие группы программистов сообщали, что разработка ускорилась, проекты стали чаще завершаться в запланированные сроки и в рамках бюджета. Эти успехи объяснялись тем, что структурированные программы стали гораздо понятнее, их стало легче отлаживать и модифицировать, а главное, повышалась вероятность, что в них нет ошибок.

Бём и Якопини продемонстрировали, что для написания любой программы нужны всего три управляющие структуры: **последовательность (sequence)**, **выбор (selection)** и **цикл (iteration**, или **loop)**. Сейчас мы расскажем, как они реализованы в C++.

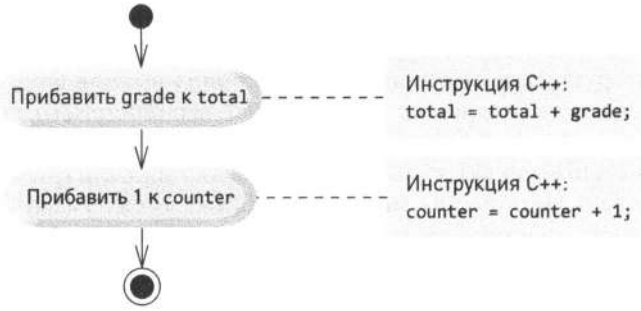
---

[1] Виктор Зверович (Victor Zverovich), «Text Formatting», 16 июля 2019 г. Был доступен: 15 сентября 2023 г. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0645r10.html.

[2] К. Бём (C. Böhm) и Дж. Якопини (G. Jacopini), «Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules», ACM, выпуск 9, № 5, май 1966 г., с. 336–371.

3.2.1. Последовательность

Последовательность — это базовая структура, встроенная в язык C++. По умолчанию (если не указано иное) все инструкции выполняются последовательно: в том порядке, как они написаны сверху вниз в исходном коде. Следующая **диаграмма действий (activity diagram)**[1] описывает простую последовательность, состоящую из двух вычислительных инструкций:



C++ позволяет вам последовательно выполнять любое количество действий. Кроме того, любое отдельное действие в программе вы можете заменить несколькими последовательными действиями.

Диаграмма действий, также называемая диаграммой деятельности, моделирует **рабочий процесс (workflow)**, происходящий в программе или в ее фрагменте. В частности, на такой диаграмме можно показать линейный алгоритм вычислений, как в нашем примере. Основа любой диаграммы действий — фигуры-символы: прямоугольники с закругленными углами (они обозначают действия), ромбы и маленькие круги. Эти фигуры соединены стрелками, обозначающими **поток управления (flow of control)**, то есть тот порядок, в котором должны выполняться действия.

Наша простая диаграмма с последовательной структурой описывает два **состояния действия (action states)**. Каждое из них содержит **выражение действия (action expression)**: «прибавить `grade` к `total`» и «прибавить `1` к `counter`» — это те конкретные действия, которые надо выполнить. Стрелками показаны три перехода между ними.

Черный круг в верхней части диаграммы обозначает **начальное состояние (initial state)**, то есть начало потока управления перед выполнением программой моделируемых действий. Белый круг с черным кругом внутри в нижней части диаграммы обозначает **конечное состояние (final state)**, то есть завершение потока управления после выполнения программой всех указанных действий.

---

[1] Мы используем UML-диаграммы (UML — Unified Modeling Language) сначала в главах 3 и 4, рассказывая о передаче управления в управляющих инструкциях, а затем в главах 9 и 10, посвященных разработке классов.

Прямоугольники с загнутыми верхними правыми углами — это **примечания (notes)**, поясняющие назначение символов на UML-диаграмме (эти примечания подобны комментариям в исходном коде C++). Каждый символ соединен со своим примечанием пунктирной линией. В нашем примере примечания показывают, как преобразовать каждое из состояний в исходный код на языке C++. Как правило, сам исходный код на таких диаграммах не пишут.

3.2.2. Инструкции выбора

В языке C++ есть три вида **инструкций выбора (selection statements)**, также называемых **инструкциями ветвления**. Инструкция `if` выполняет действие (или группу действий), если условие истинно, или не выполняет его, если условие ложно. Инструкция `if...else` выполняет действие (или группу действий), если условие истинно, или выполняет альтернативное действие (или группу действий), если условие ложно. Инструкция `switch` (она обсуждается в главе 4) выполняет одно из нескольких действий (или одну из групп действий) в зависимости от значения выражения.

Инструкция `if` называется **инструкцией одиночного выбора (single-selection statement)**, поскольку она выполняет или игнорирует действие (или группу действий). Инструкция `if...else` называется **инструкцией двойного выбора (double-selection statement)**, поскольку она выбирает между двумя различными действиями (или группами действий). Инструкция `switch` называется **инструкцией множественного выбора (multiple-selection statement)**, поскольку она выбирает среди множества действий (или групп действий).

3.2.3. Инструкции цикла

В языке C++ есть четыре вида **инструкций цикла (looping statements)**, также называемых **инструкциями повторения (repetition statements)** или **инструкциями итерации**[1] **(iteration statements)**: `while`, `do...while`, `for` и `for` для диапазона значений. Все они многократно выполняют действие (или группу действий) до тех пор, пока остается истинным **условие продолжения цикла (loop-continuation condition)**. Инструкции `while` и `for` выполняют свое действие (или группу действий) ноль или более раз; если условие продолжения с самого начала является ложным (`false`), то действие (или группа действий) не выполняется. Инструкция `do...while` выполняет свое действие (или группу действий) один или несколько раз. В главе 4 мы рассмотрим инструкции `do...while` и `for`, а в главе 6 — инструкцию `for` для диапазона значений.

Ключевые слова

Слова `if`, `else`, `switch`, `while`, `do` и `for` относятся к ключевым (зарезервированным) словам языка C++. Ключевые слова не могут использоваться в качестве иден-

[1] Итерация — повторяющееся действие, проход цикла, шаг циклического процесса; обработка данных в цикле. — *Примеч. ред.*

тификаторов (имен переменных) и содержат только строчные буквы, а иногда еще и символы подчеркивания. В следующей таблице приведен полный список ключевых слов C++:

Ключевые слова C++				
alignas	alignof	and	and_eq	asm
auto	bitand	bitor	bool	break
case	catch	char	char16_t	char32_t
class	compl	const	const_cast	constexpr
continue	decltype	default	delete	do
double	dynamic_cast	else	enum	explicit
export	extern	false	final	float
for	friend	goto	if	import
inline	int	long	module	mutable
namespace	new	noexcept	not	not_eq
nullptr	operator	or	or_eq	override
private	protected	public	register	reinterpret_cast
return	short	signed	sizeof	static
static_assert	static_cast	struct	switch	template
this	thread_local	throw	true	try
typedef	typeid	typename	union	unsigned
using	void	volatile	virtual	wchar_t
while	xor	xor_eq		
Ключевые слова, введенные в C++20				
char8_t	concept	consteval	constinit	co_await
co_return	co_yield	requires		

3.2.4. Комбинации управляющих инструкций

Итак, в языке C++ есть только три вида управляющих структур, которые мы далее будем называть **управляющими инструкциями (control statements)**:

- последовательность;
- выбор (`if`, `if...else` и `switch`);
- цикл (`while`, `do...while`, `for` и `for` для диапазона значений).

Создавая любую программу, вы комбинируете эти инструкции в соответствии с тем алгоритмом, который реализуете. Каждую управляющую инструкцию можно смоделировать в виде диаграммы действий. Такая диаграмма будет содержать начальное и конечное состояния, то есть точку входа и точку выхода этой инструк-

ции. Управляющие инструкции с **одной точкой входа и одной точкой выхода (single-entry/single-exit control statements)** упрощают создание простых для понимания программ: вы просто соединяете точку выхода со следующей точкой входа, составляя таким образом **последовательность управляющих инструкций (control-statement stacking)**. Второй способ соединения управляющих инструкций — это **вложенность (nesting)**: вы помещаете одну инструкцию внутрь другой. Таким образом, все алгоритмы в программах на языке C++ составляются только из трех видов управляющих инструкций, соединяемых лишь двумя способами. Такова суть простоты этого языка.

3.3. Инструкция одиночного выбора if

Вы познакомились с инструкцией `if` в разделе 2.6. Она применяется для выбора между двумя вариантами последовательности действий. Например, предположим, что проходной балл на экзамене равен `60`. Следующая инструкция C++ определяет, является ли условие `studentGrade >= 60` истинным:

```
if (studentGrade >= 60) {
   cout << "Passed";
}
```

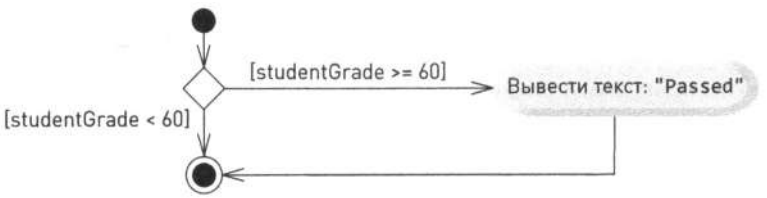
Если условие истинно, выводится сообщение `"Passed"` и выполняется следующая по порядку инструкция. Если условие ложно, инструкция вывода игнорируется и выполняется следующая по порядку инструкция. Отступ во второй строке этой инструкции необязателен, но рекомендуется для наглядности кода.

Логический тип данных (bool)

В главе 2 вы формировали условия с помощью операторов равенства и сравнения. В качестве условий также можно использовать любые выражения, результатом которых может быть ноль. При этом ноль означает, что условие ложно, а любое ненулевое значение — что условие истинно. Также в C++ есть логический тип данных (тип `bool`). Логические переменные (переменные типа `bool`) могут принимать только два значения: `true` (истинно) и `false` (ложно). Слова `true` и `false` являются в C++ ключевыми. Компилятор может неявно преобразовывать `true` в `1`, а `false` — в `0`.

Диаграмма действий для инструкции if

Следующая диаграмма показывает поток управления, соответствующий инструкции `if`.



Эта диаграмма содержит очень важный символ: ромб — **символ принятия решения (decision symbol)**, обозначающий альтернативность дальнейших действий. После ромба поток управления продолжается в зависимости от **сторожевых условий (guard conditions)** — они указаны в квадратных скобках рядом со стрелками перехода, выходящими из ромба. Если условие истинно, поток управления продолжается с того действия, на которое направлена соответствующая стрелка перехода. Диаграмма показывает: если значение переменной `studentGrade` больше или равно `60` (если это условие истинно), то программа выводит сообщение `"Passed"` и затем переходит в конечное состояние. Если же `studentGrade` меньше `60`, то программа немедленно переходит в конечное состояние без вывода сообщения. Инструкция `if` — это управляющая инструкция с одним входом и одним выходом.

3.4. Инструкция двойного выбора if...else

Итак, инструкция выбора `if` выполняет действие, только если условие истинно. В отличие от нее, инструкция двойного выбора `if...else` позволяет вам указать действие, которое будет выполнено при истинном условии, а также альтернативное действие, которое будет выполнено при ложном условии. Например, следующая инструкция выводит сообщение `"Passed"`, если `studentGrade >= 60`, или сообщение `"Failed"`, если значение этой переменной меньше `60`:

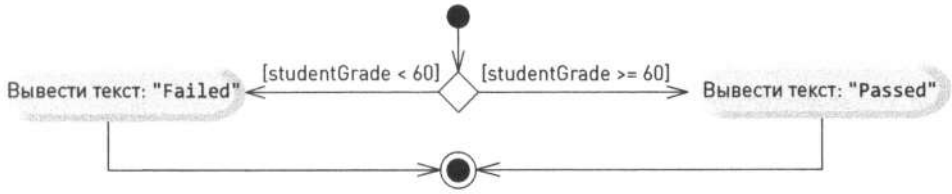
```
if (studentGrade >= 60) {
   cout << "Passed";
}
else {
   cout << "Failed";
}
```

При любом значении переменной `studentGrade` после вывода соответствующего сообщения будет выполнена следующая инструкция в последовательности.

Обратите внимание, что тело `else` мы тоже записали с отступом шириной в три символа. Независимо от выбранной вами ширины отступа, она должна быть одинаковой во всей вашей программе.

Диаграмма действий для инструкции if...else

Следующая диаграмма показывает поток управления, соответствующий инструкции `if...else`:



3.4.1. Вложенные инструкции if...else

Чтобы программа могла выбирать между многими вариантами действий, можно использовать вложенные инструкции `if...else`, то есть поместить их друг в друга по принципу матрешки. Например, следующая структура из вложенных инструкций `if...else` выводит одно из пяти сообщений: `"A"` — для оценок, которые больше или равны `90`, `"B"` — для оценок от `80` до `89`, `"C"` — для оценок от `70` до `79`, `"D"` — для оценок от `60` до `69` и `"F"` — для всех остальных оценок. Мы обозначили четыре уровня вложенности четырьмя прямоугольниками.

```
if (studentGrade >= 90) {
   cout << "A";
}
else {
   if (studentGrade >= 80) {
      cout << "B";
   }
   else {
      if (studentGrade >= 70) {
         cout << "C";
      }
      else {
         if (studentGrade >= 60) {
            cout << "D";
         }
         else {
            cout << "F";
         }
      }
   }
}
```

Если переменная `StudentGrade` больше или равна `90`, то первые четыре условия во вложенных инструкциях `if...else` будут истинными, но выполнится только тело секции `if` в первой из них («самой внешней»), а тело соответствующей секции `else` будет пропущено. Всю эту управляющую структуру можно записать и другим способом, с меньшим количеством фигурных скобок и отступов, при этом программа выполнит точно такие же действия:

```
if (studentGrade >= 90) {
   cout << "A";
}
else if (studentGrade >= 80) {
   cout << "B";
}
else if (studentGrade >= 70) {
```

```
   cout << "C";
}
else if (studentGrade >= 60) {
   cout << "D";
}
else {
   cout << "F";
}
```

Такая запись позволяет избежать больших сдвигов строк вправо, которые могут приводить к неудобным переносам строк на экране и в распечатках. Обратите внимание, что здесь и в последующих примерах мы обязательно заключаем тело каждой управляющей инструкции в фигурные скобки ({ и }), чтобы избежать логической ошибки, называемой «висячие `else`».

3.4.2. Блоки

Инструкция `if` предполагает, что ее тело состоит только из одной инструкции. Чтобы включить несколько последовательных инструкций в секцию `if` или `else`, поместите их между фигурными скобками. Опытные программисты рекомендуют всегда заключать в фигурные скобки даже единственную инструкцию в секции `if` или `else`. Последовательность инструкций между фигурными скобками (например, в теле управляющей инструкции или функции) образует **блок (block)**. В любом месте функции, где допустима отдельная инструкция, ее можно заменить блоком.

В следующем примере показан блок из двух инструкций в теле секции `else`:

```
if (studentGrade >= 60) {
   cout << "Passed";
}
else {
   cout << "Failed\n";
   cout << "You must retake this course.";
}
```

Если `StudentGrade` меньше `60`, программа выполнит обе инструкции в теле `else` и выведет сообщение:

```
Failed
You must retake this course.
```

Что будет, если в этом примере убрать (забыть) фигурные скобки в теле секции `else`? Тогда эта инструкция окажется вне тела `else`:

```
cout << "You must retake this course.";
```

Из-за этого она будет выполняться всегда, независимо от значения `StudentGrade`. Возникнет логическая ошибка.

Пустая инструкция

Любую отдельную инструкцию можно заменить не только блоком, но и **пустой инструкцией (empty statement)**, которая состоит только из точки с запятой (`;`). Пустая инструкция не влияет на выполнение программы.

3.4.3. Тернарный условный оператор ?:

В языке C++ есть **условный оператор (conditional operator)** `?:`, который вы можете использовать вместо инструкции `if...else`, чтобы сделать код короче и понятнее. Это единственный оператор C++, принимающий три операнда, поэтому его называют **тернарным оператором (ternary operator)** — от латинского слова «ternarius» — «тройной». Операнды вместе с символами `?` и `:` образуют **условное выражение (conditional expression)**. Например, следующая инструкция выводит значение условного выражения:

```
cout << (studentGrade >= 60 ? "Passed" : "Failed");
```

Операнд слева от символа `?` — это условие. Второй операнд (находящийся между символами `?` и `:`) — это значение выражения в том случае, если условие истинно. Операнд справа от символа `:` — это значение выражения в том случае, если условие ложно. Условное выражение в этой инструкции вычисляется как строка `"Passed"`, если условие

```
studentGrade >= 60
```

имеет значение `true`, или вычисляется как строка `"Failed"`, если это условие имеет значение `false`. Таким образом, условный оператор `?:` выполняет те же действия, что и инструкция `if...else` из нашего первого примера в разделе 3.4. У этого оператора низкий приоритет, поэтому такое условное выражение обычно помещают в круглые скобки.

3.5. Инструкция цикла while

Инструкция цикла позволяет вам указать, что программа должна повторять какое-либо действие до тех пор, пока заданное условие остается истинным.

В качестве примера имеющейся в C++ инструкции цикла `while` рассмотрим код, вычисляющий наименьшее значение числа `3` в натуральной степени, которое больше `100`. После выполнения цикла `while` результат будет сохранен в переменной `product`:

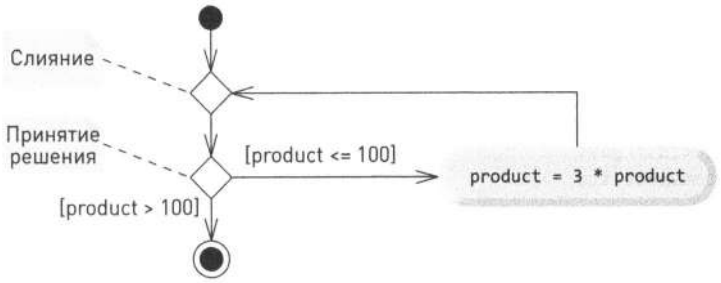
```
int product{3};
while (product <= 100) {
   product = 3 * product;
}
```

При каждом выполнении тела цикла `while` переменная `product` умножается на `3`, то есть последовательно принимает значения `9`, `27`, `81` и `243`. Когда значение этой переменной достигнет `243`, условие `product <= 100` станет ложным и цикл

завершится. При этом в переменной `product` останется значение `243`, после чего выполнение программы продолжится со следующей инструкции.

Диаграмма действий для инструкции while

На следующей UML-диаграмме, моделирующей приведенный выше фрагмент кода с инструкцией `while`, вы видите **символ слияния (merge symbol)**:



Символы слияния на UML-диаграммах имеют вид ромбов, то есть выглядят так же, как символы принятия решений. Символ слияния объединяет два потока управления в один. На нашей диаграмме символ слияния объединяет два перехода: из начального состояния и из состояния действия. После каждого из этих переходов следует переход к символу принятия решения о том, надо ли начинать (или продолжать) выполнение цикла.

Как отличать символы принятия решений от символов слияния? По количеству входящих и исходящих стрелок перехода. Символ принятия решения имеет одну входящую стрелку и несколько исходящих — они обозначают возможные варианты переходов. При этом около каждой исходящей стрелки должно быть указано условие перехода. Символ слияния, наоборот, имеет несколько входящих стрелок и одну исходящую — она обозначает объединенный поток управления после слияния. При этом у любых стрелок перехода, соединенных с символом слияния, никогда не бывает указано условие перехода.

3.6. Цикл со счетчиком

Рассмотрим следующую задачу:

> *Класс из 10 учеников сделал контрольную работу. Полученные учениками оценки (целые числа в диапазоне от 0 до 100) вводятся с клавиатуры. Требуется найти среднюю арифметическую оценку по классу.*

Чтобы найти среднюю арифметическую оценку, надо сумму оценок разделить на количество учеников. Программа должна по очереди получить каждую оценку, суммировать их все, вычислить среднюю оценку и вывести результат.

Мы будем использовать **цикл со счетчиком (counter-controlled iteration)** для ввода оценок с клавиатуры по очереди. Этот метод использует счетчик для контроля количества выполненных блоков инструкций. В нашем примере цикл завершится, когда значение счетчика превысит `10`.

3.6.1. Реализация цикла со счетчиком

На илл. 3.1 показано, как функция `main` вычисляет среднюю оценку с помощью цикла со счетчиком. Она запрашивает у пользователя десять оценок, после чего вычисляет и выводит на экран результат.

```
// fig03_01.cpp
// Вычисление средней оценки с помощью цикла со счетчиком
#include <iostream>
using namespace std;

int main() {
   // Инициализируем переменные
   int total{0}; // Сумма оценок
   int gradeCounter{1}; // Счетчик оценок

   // Обрабатываем данные в цикле со счетчиком
   while (gradeCounter <= 10) { // Выполняем 10 итераций
      cout << "Enter grade: "; // Выводим сообщение
      int grade;
      cin >> grade; // Вводим оценку
      total = total + grade; // Прибавляем оценку к сумме оценок
      gradeCounter = gradeCounter + 1; // Увеличиваем счетчик на 1
   }

   // Ввод данных завершен
   int average{total / 10}; // Деление чисел типа int дает результат типа int

   // Выводим сумму оценок и среднюю оценку
   cout << "\nTotal of all 10 grades is " << total;
   cout << "\nClass average is " << average << "\n";
}
```

```
Enter grade: 67
Enter grade: 78
Enter grade: 89
Enter grade: 67
Enter grade: 87
Enter grade: 98
Enter grade: 93
Enter grade: 85
Enter grade: 82
Enter grade: 100
Total of all 10 grades is 846
Class average is 84
```

Илл. 3.1 | Вычисление средней оценки с помощью цикла со счетчиком

Локальные переменные в функции main

В строках 8, 9, 14 и 21 объявляются целочисленные переменные `total`, `gradeCounter`, `grade` и `average`. Переменная `grade` хранит введенное пользователем число. Это **локальная переменная (local variable)**. Локальной называется переменная, объявленная в блоке (например, в теле функции); ее можно использовать только от момента объявления до закрывающей фигурной скобки блока. Объявление локальной переменной должно предшествовать первому обращению к ней. Переменную `grade`, объявленную в цикле `while`, можно использовать только в пределах тела этого цикла.

Инициализация переменных total и gradeCounter

В строках 8–9 объявляются и инициализируются две переменные: `total` со значением `0` и `gradeCounter` со значением `1`. Инициализация происходит до первого использования этих переменных в вычислениях.

Ввод десяти оценок пользователем

Инструкция цикла `while` (строки 12–18) продолжает выполняться до тех пор, пока значение переменной `gradeCounter` меньше или равно `10`. В строке 13 выводится сообщение `"Enter grade: "`, а в строке 15 введенное пользователем число присваивается переменной `grade`. В строке 16 это число суммируется с переменной `total`, и результат сохраняется в `total`, заменяя предыдущее значение этой переменной. В строке 17 к переменной `gradeCounter` прибавляется `1`, что означает завершение обработки очередной оценки и готовность к следующему повторению цикла. Увеличение `gradeCounter` в итоге приведет к превышению максимально допустимого значения в условии цикла (`10`), и тогда цикл завершится.

Вычисление и вывод среднего арифметического

После завершения цикла в строке 21 выполняется вычисление средней арифметической оценки по классу. Это происходит при инициализации переменной `average`. В строке 24 на экран выводится сообщение `"Total of all 10 grades is "`, за которым следует значение переменной `total`. В строке 25 выводится сообщение `"Class average is "` и средняя оценка. В строке 26 программа завершается.

3.6.2. Целочисленное деление с усечением

Вычисление средней арифметической оценки в этом примере дает целочисленный результат. Сумма введенных оценок равна `846`, и в обычной арифметике после деления на `10` получилось бы `84.6`. Такие числа, содержащие десятичные точки, в C++ называются **числами с плавающей точкой (floating-point numbers)**. Но в нашей программе результатом выражения `total / 10` будет целое число `84`, поскольку оба операнда имеют тип `int`. Деление двух чисел типа `int` является **целочисленным делением (integer division)**, то есть любая дробная часть результата отбрасывается. Например, результатом вычисления выражения `7 / 4` в обычной арифметике будет `1.75`. Но при целочисленном делении результат усекается до `1` вместо округления

до 2. В следующем разделе мы покажем, как получить в результате аналогичных вычислений число с плавающей точкой.

3.7. Цикл с контрольным значением

Давайте обобщим задачу, сформулированную в разделе 3.6. Новое требование будет таким:

> *Разработайте аналогичную программу вычисления средней арифметической оценки, которая будет работать при любом количестве учеников в классе.*

В предыдущем примере в условии задачи было задано количество учеников, то есть количество вводимых оценок (`10`) было известно программе заранее. А теперь мы не знаем, сколько оценок введет пользователь во время выполнения программы. Программа должна правильно обрабатывать произвольное количество оценок.

Один из способов решения этой задачи — использование **контрольного значения (sentinel value)**, также называемого **сигнальным (signal value)**, **фиктивным (dummy value)** или **флаговым (flag value)** значением. Контрольное значение указывает на окончание ввода данных. Пользователь вводит оценки до тех пор, пока не введет их все. Затем он вводит заранее известное контрольное значение, которое означает, что оценки больше вводиться не будут.

Вы должны выбрать такое контрольное значение, которое нельзя перепутать с любыми возможными оценками. Оценки за контрольную работу — это неотрицательные целые числа, поэтому в качестве контрольного значения хорошо подойдет отрицательное число `-1`. В этом случае поток входных данных, которые должна обработать программа, будет, например, таким: `95`, `96`, `75`, `74`, `89` и, наконец, `-1`. Затем программа должна будет вычислить и вывести среднее арифметическое значение для оценок `95`, `96`, `75`, `74` и `89`, поскольку значение `-1` является контрольным и не должно участвовать в расчетах.

Не исключено, что пользователь введет `-1` в самом начале, то есть количество оценок будет равно `0`. Мы должны учесть это перед началом вычислений. Согласно стандарту C++, результат деления на ноль в выражениях с плавающей точкой **не определен (undefined)**. Поэтому перед любой операцией деления (`/`) или вычисления остатка от деления (`%`), в которой правый операнд может быть равен нулю, надо проверить значение правого операнда и в случае нуля обработать это состояние (например, вывести сообщение об ошибке), не разрешая программе приступить к недопустимому делению на ноль.

3.7.1. Реализация цикла с контрольным значением

На илл. 3.2 показан пример цикла с контрольным значением. Предполагается, что все оценки, введенные пользователем, будут целыми числами, но на этот раз мы получим их среднее арифметическое в виде числа с плавающей точкой. Такие

числа нельзя хранить в переменных типа `int`. Для чисел с плавающей точкой в языке C++ есть типы данных `float`, `double` и `long double`. Основное различие между ними — это **точность (precision)**. По сравнению с типом `float` тип `double` обеспечивает значительно большую точность вычислений (то есть позволяет работать с числами намного большей величины или с намного большим количеством цифр после точки). В свою очередь, тип `long double` обеспечивает значительно большую точность вычислений по сравнению с типом `double`. В главе 4 мы подробнее расскажем обо всех типах переменных, предназначенных для чисел с плавающей точкой.

```
// fig03_02.cpp
// Вычисление средней оценки с помощью цикла с контрольным значением
#include <iostream>
#include <iomanip> // Заголовок для параметризованных манипуляторов потока
using namespace std;

int main() {
   // Инициализируем переменные
   int total{0}; // Сумма оценок
   int gradeCounter{0}; // Счетчик оценок

   // Обрабатываем данные
   // Выводим сообщение и вводим оценку
   cout << "Enter grade or -1 to quit: ";
   int grade;
   cin >> grade;

   // Повторяем цикл до ввода контрольного значения
   while (grade != -1) {
      total = total + grade; // Прибавляем оценку к сумме оценок
      gradeCounter = gradeCounter + 1; // Увеличиваем счетчик

      // Выводим сообщение и вводим следующую оценку
      cout << "Enter grade or -1 to quit: ";
      cin >> grade;
   }

   // Ввод данных завершен
   // Если введена хотя бы одна оценка...
   if (gradeCounter != 0) {
      // Используем тип double для вычисления среднего арифметического
      double average{static_cast<double>(total) / gradeCounter};

      // Выводим сумму оценок и среднюю оценку (с двумя цифрами после точки)
      cout << "\nTotal of the " << gradeCounter
         << " grades entered is " << total;
      cout << setprecision(2) << fixed;
      cout << "\nClass average is " << average << "\n";
   }
   else { // Ни одна оценка не введена, выводим сообщение об этом
      cout << "No grades were entered\n";
```

Окончание ⇨

```
42    }
43 }
```

```
Enter grade or -1 to quit: 97
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 72
Enter grade or -1 to quit: -1
Total of the 3 grades entered is 257
Class average is 85.67
```

Илл. 3.2 | Вычисление средней оценки с помощью цикла с контрольным значением

Напомним, что целочисленное деление дает целочисленный результат. Но в данном случае мы включили в программу `static_cast` — один из **операторов преобразования типов (cast operators)**, также называемых **операторами приведения типов**. Он означает, что результатом деления должно быть число с плавающей точкой. В программе последовательно выполняются две управляющие инструкции: сначала `while`, а затем `if...else`. Значительная часть кода аналогична коду предыдущей программы (илл. 3.1), поэтому мы поясним только основные различия.

Различия в логике работы цикла с контрольным значением и цикла со счетчиком

В строке 10 счетчик оценок инициализируется значением `0`, поскольку ни одна оценка еще не введена. Не забывайте, что программа использует для ввода оценок цикл с контрольным значением. Значение `gradeCounter` увеличивается на единицу, когда пользователь вводит оценку. В строке 32 объявляется переменная `average` типа `double`, в которой полученная средняя арифметическая оценка по классу сохраняется в виде числа с плавающей точкой.

Сравните логику программ, показанных на илл. 3.1 и 3.2. В цикле со счетчиком инструкция `while` (строки 12–18) выполняется ровно `10` раз, и при каждом ее выполнении программа получает от пользователя очередную оценку. А в цикле с контрольным значением программа запрашивает и получает первое значение (строки 14 и 16 на илл. 3.2) еще до перехода к циклу `while`. От первого введенного значения зависит, состоится ли переход к телу цикла `while`. Если условие в строке 19 ложно (если пользователь сразу ввел контрольное значение `-1`), то программа не получит ни одной оценки и тело цикла не выполнится ни одного раза. Если же это условие истинно, начинается выполнение тела цикла: переменная `grade` суммируется с `total` и увеличивается счетчик `gradeCounter`. Затем в строках 24–25 программа запрашивает у пользователя следующее значение, после чего достигается закрывающаяся фигурная скобка цикла в строке 26 и выполнение продолжается с проверки условия цикла `while` в строке 19. При этой проверке используется последнее на данный момент значение переменной `grade`, введенное пользователем. От него зависит, будет ли тело цикла выполнено снова.

Каждое значение переменной `grade` пользователь вводит непосредственно перед проверкой условия продолжения цикла. Это позволяет программе определить

перед дальнейшими действиями, является ли введенное значение контрольным (надо ли прибавлять `grade` к `total`). Если введено контрольное значение, цикл завершается и программа не прибавляет `-1` к сумме оценок.

После завершения цикла выполняется инструкция `if...else` в строках 30–42. Условие в строке 30 определяет, была ли введена хотя бы одна оценка. Если пользователь не ввел ни одной оценки, выполняется тело `else` инструкции `if...else`, то есть выводится сообщение `"No grades were entered"`. После инструкции `if...else` программа завершается.

3.7.2. Явное и неявное преобразование основных типов данных

Продолжим пояснения к коду, показанному на илл. 3.2. Если пользователь ввел хотя бы одну оценку, то в строке 32 вычисляется среднее арифметическое суммы чисел:

```
double average{static_cast<double>(total) / gradeCounter};
```

Напомним, что целочисленное деление всегда дает целочисленный результат, даже если вы записываете его в переменную типа `double`. Например, следующая инструкция отбросит (удалит) дробную часть результата деления еще до того, как этот результат будет записан в переменную `average` при ее инициализации:

```
double average{total / gradeCounter};
```

Оператор static_cast

Чтобы при делении целых чисел получить результат с плавающей точкой, надо сначала создать временное значение с плавающей точкой при помощи оператора `static_cast`. В строке 32 создается временная копия указанной в круглых скобках переменной `total`, приведенная к типу `double`, указанному в угловых скобках. При этом сама переменная `total` не меняется, ее значение остается целочисленным. Такое использование оператора `static_cast` называется **явным преобразованием типов (explicit type conversion)**. Кроме оператора `static_cast`, вы встретитесь в книге и с другими операторами преобразования типов. Преобразование типов также называют **приведением типов (type casting)**.

Продвижение типов

Итак, среднее арифметическое будет получено делением значения временной копии переменной `total` (имеющей тип `double`) на значение переменной `gradeCounter` (имеющей тип `int`). Но компилятор C++ может вычислять только те арифметические выражения, в которых все операнды имеют один и тот же тип. Поэтому если типы операндов отличаются, компилятор автоматически выполняет **продвижение типов (type promotion)**, также называемое **неявным преобразованием типов (implicit type conversion)**. В выражениях с операндами `int` и `double` компилятор преобразует целочисленные операнды (`int`) в числа с плавающей

точкой (`double`). В строке 32 компилятор преобразует тип временной копии переменной `gradeCounter` в тип `double`, а затем выполняет деление. После этого переменная `average` инициализируется результатом деления, имеющим тип `double`. В разделе 5.6 мы обсудим разрешенные в языке C++ неявные преобразования основных типов данных.

Возможности операторов приведения типов

Операторы приведения типов можно использовать со всеми основными типами данных, а также с другими типами, как вы увидите в главе 9. Просто укажите нужный тип в угловых скобках (`<` и `>`) после ключевого слова `static_cast`. Это **унарный оператор (unary operator)** — у него есть только один операнд. В языке C++ есть и другие унарные операторы, например `+` (плюс) и `-` (минус), применяемые в таких выражениях, как `+5` или `-7`. Операторы приведения типов имеют второй приоритет.

3.7.3. Форматирование чисел с плавающей точкой

Сейчас мы кратко обсудим средства форматирования результата вычислений, использованные в программе вычисления средней оценки (илл. 3.2).

Параметризованный манипулятор потока setprecision

В строке 37 вызывается **манипулятор потока (stream manipulator)** `setprecision` с параметром `2`: `setprecision(2)`. Он указывает компилятору, что числа с плавающей точкой надо выводить в поток с заданной **точностью (precision)**: с двумя цифрами после точки (например, `92.37`). Этот манипулятор является **параметризованным (parameterized)**, поскольку для выполнения задачи ему нужен параметр — количество цифр после точки (в данном случае `2`). В любой программе, использующей параметризованные манипуляторы потока, должен быть включен заголовок `<iomanip>` (строка 4).

Непараметризованный манипулятор потока fixed

Непараметризованный (non-parameterized), то есть не имеющий параметра, манипулятор потока `fixed` в строке 37 указывает, что числа с плавающей точкой надо выводить в поток в **формате с фиксированной точкой (fixed-point format)**. Это альтернатива **научной записи (scientific notation)**, в которой любое число выглядит в виде мантиссы (числа в диапазоне от `1.0` до `10.0`) и порядка (степени числа `10`). Например, число `3100.0` (три тысячи сто целых ноль десятых) в научной записи отображается так: 3.1e+03 (то есть $3.1 * 10^3$). Этот формат особенно удобен для очень больших и очень маленьких чисел.

В формате с фиксированной точкой, в отличие от научной записи, числа выглядят привычнее. И даже если число целое, после него всегда отображается десятичная точка, а затем нули (например, `88.00`). Для сравнения, если не включить форматирование с фиксированной точкой, это же число будет выведено без точки и нулей: `88`.

Манипуляторы потока `setprecision` и `fixed` относятся к **липким настройкам (sticky settings)**. Это значит, что после однократного выполнения инструкции с таким манипулятором он «приклеивается» к вашей программе: все числа с плавающей точкой с этого момента будут форматироваться в соответствии с заданными настройками, пока вы их не измените.

Округление чисел с плавающей точкой

Когда используются манипуляторы потока `fixed` и `setprecision`, выводимое значение переменной округляется до нужного вам количества десятичных знаков, то есть до той точности, которую вы указали. Но само значение этой переменной в памяти остается неизменным. Например, при точности с двумя цифрами после точки числа `87.946` и `67.543` округляются до `87.95` и `67.54` соответственно[1].

Итак, в строках 37–38 на илл. 3.2 выводится среднее арифметическое значение, округленное до сотых и содержащее ровно две цифры справа от десятичной точки. Три оценки, введенные во время выполнения программы (`97`, `88` и `72`), в сумме составляют `257`, что дает среднее значение `85.666...` и отображается в виде округленного числа `85.67`.

3.8. Вложенные управляющие инструкции

Мы показали, что управляющие инструкции могут выполняться последовательно. В этом разделе мы рассмотрим второй способ (из двух возможных) структурированного соединения управляющих инструкций: **вложенность (nesting)** одной управляющей инструкции в другую.

3.8.1. Постановка задачи

Рассмотрим такую задачу:

> *Колледж предлагает курс подготовки студентов к экзамену в университет. В прошлом году десять студентов, окончивших этот курс, сдавали экзамен. Колледж хочет знать, насколько хорошо его студенты сдали экзамен. Напишите программу для анализа результатов. У вас есть список из десяти студентов. Рядом с каждым именем стоит число 1, если студент сдал экзамен, и число 2, если не сдал.*
>
> *Программа должна работать так:*
>
> 1. *Получить каждый результат (1 или 2), выводя на экран сообщение «Enter result» при каждом запросе результата.*
> 2. *Подсчитать количество результатов каждого типа.*

[1] Если из кода, показанного на илл. 3.2, убрать манипуляторы setprecision и fixed, компилятор C++ установит по умолчанию точность с четырьмя цифрами, а если оставить только манипулятор fixed, — с шестью.

3. *Вывести сводку результатов, указав количество студентов, сдавших и не сдавших экзамен.*
4. *Если экзамен сдали более восьми студентов, вывести сообщение «Bonus to instructor!».*

3.8.2. Реализация программы

На илл. 3.3 вы видите код этой программы, использующий цикл со счетчиком, а также примеры ввода и вывода данных. В строках 8–10 и 16 объявляются переменные, которые нужны для обработки результатов экзамена. Цикл `while` (строки 13–29) повторяется `10` раз. В каждой итерации цикла программа получает и обрабатывает один результат экзамена. Обратите внимание, что инструкция `if...else` (строки 20–25), обрабатывающая каждый результат, вложена в цикл `while`. Если значение переменной `result` равно `1`, инструкция `if...else` увеличивает счетчик `passes`, в противном случае инструкция увеличивает `failures` (предполагая, что введено число 2)[1]. В строке 28 счетчик `studentCounter` увеличивается перед повторной проверкой условия цикла в строке 13.

```
 1 // fig03_03.cpp
 2 // Анализ результатов экзамена с помощью вложенных управляющих инструкций
 3 #include <iostream>
 4 using namespace std;
 5
 6 int main() {
 7    // Инициализируем переменные при объявлении
 8    int passes{0};
 9    int failures{0};
10    int studentCounter{1};
11
12    // Обрабатываем данные о 10 студентах в цикле со счетчиком
13    while (studentCounter <= 10) {
14       // Выводим сообщение и вводим результат экзамена
15       cout << "Enter result (1 = pass, 2 = fail): ";
16       int result;
17       cin >> result;
18
19       // В тело цикла while вложена инструкция if...else
20       if (result == 1) {
21          passes = passes + 1;
22       }
23       else {
24          failures = failures + 1;
25       }
26
27       // Увеличиваем studentCounter на 1 (поэтому цикл в итоге завершится)
```

[1] Предположение, что результат равен 2, может быть ошибочным, если пользователь ввел некорректные данные. Позже мы обсудим методы проверки вводимых данных.

```
28        studentCounter = studentCounter + 1;
29     }
30
31     // Ввод данных завершен; выводим результаты
32     cout << "Passed: " << passes << "\nFailed: " << failures << "\n";
33
34     // Проверяем, сдали ли экзамен более восьми студентов
35     if (passes > 8) {
36        cout << "Bonus to instructor!\n";
37     }
38 }
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 9
Failed: 1
Bonus to instructor!
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 6
Failed: 4
```

Илл. 3.3 | Анализ результатов экзамена с помощью вложенных управляющих инструкций

После ввода десяти значений цикл завершается и в строке 32 выводится количество сданных и проваленных экзаменов. Инструкция `if` в строках 35–37 проверяет, сдали ли экзамен более восьми студентов, и если это так, выводит сообщение `"Bonus to instructor!"`.

На илл. 3.3 показаны примеры работы программы с двумя наборами данных. В первом случае условие в строке 35 истинно: более восьми студентов сдали экзамен, поэтому программа выводит сообщение о поощрении преподавателя.

3.8.3. Предотвращение сужающих преобразований с помощью скобочной инициализации

Рассмотрим скобочную инициализацию переменной в строке 10:

```
int studentCounter{1};
```

Напомним, что этот синтаксис появился в C++11. В более ранних версиях языка аналогичная инструкция записывалась так:

```
int studentCounter = 1;
```

Для переменных фундаментальных типов скобочная инициализация предотвращает нежелательные **сужающие преобразования (narrowing conversions)**, которые могут приводить к потере данных. Например, следующая инструкция объявления переменной пытается присвоить целочисленной переменной `x` значение `12.7` типа `double`:

```
int x = 12.7;
```

В этом случае компилятор C++ преобразует `double` в `int`, отсекая (отбрасывая) часть с плавающей точкой (`.7`). Это называется сужающим преобразованием. Оно искажает данные: переменная `x` получает неправильное значение `12`. Компиляторы обычно выдают предупреждение, но все-таки «соглашаются» компилировать такой код. Для сравнения, скобочная инициализация:

```
int x{12.7};
```

приведет к **ошибке компиляции (compilation error)**, то есть поможет вам сразу предотвратить логическую ошибку, которую потом будет трудно обнаружить. Даже если вы укажете в фигурных скобках целое число с плавающей точкой (например, `12.0`), все равно возникнет ошибка компиляции: не из-за самого числа (`12.0`), а из-за его типа (`double`).

В документации стандарта C++ не указана формулировка сообщения об ошибке компиляции. В приведенном примере сообщение Apple Xcode будет таким:

```
Type 'double' cannot be narrowed to 'int' in initializer list
```

Аналогичное сообщение Visual Studio:

```
conversion from 'double' to 'int' requires a narrowing conversion
```

И аналогичное сообщение GNU g++:

```
type 'double' cannot be narrowed to 'int' in initializer list [-Wc++11-narrowing]
```

В последующих главах мы обсудим другие особенности скобочной инициализации.

Пример из нашей прежней программы (илл. 3.1)

В строке 21 кода, показанного на илл. 3.1, есть такая инструкция:

```
int average{total / 10}; // Деление чисел типа int дает результат типа int
```

Вам может показаться, что она содержит сужающее преобразование. Но на самом деле и переменная `total`, и число `10` имеют тип `int`, поэтому и инициализирующее значение в фигурных скобках тоже имеет тип `int`. Если бы мы ранее объявили переменную `total` типа `double` или написали бы здесь `10.0` вместо `10`, тогда инициализирующее значение имело бы тип `double`, что привело бы к сообщению об ошибке компиляции из-за сужающего преобразования.

3.9. Составные операторы присваивания

Вы можете сократить следующую инструкцию:

```
c = c + 3;
```

с помощью составного оператора `+=`, который называется **оператором сложения с присваиванием (addition assignment operator)**:

```
c += 3;
```

Оператор `+=` прибавляет значение выражения, указанного справа от него, к переменной, указанной слева от него, и сохраняет результат в этой же переменной. Таким образом, выражение `c += 3` увеличивает переменную `c` на `3`. В следующей таблице показаны все арифметические составные операторы присваивания, примеры выражений и результатов вычислений с этими операторами:

Оператор	Пример выражения	Пояснение	Присваивание
Исходные значения пяти переменных типа `int`: `c = 3, d = 5, e = 4, f = 6, g = 12`			
`+=`	`c += 7`	`c = c + 7`	Значение `10` присваивается переменной `c`
`-=`	`d -= 4`	`d = d - 4`	Значение `1` присваивается переменной `d`
`*=`	`e *= 5`	`e = e * 5`	Значение `20` присваивается переменной `e`
`/=`	`f /= 3`	`f = f / 3`	Значение `2` присваивается переменной `f`
`%=`	`g %= 9`	`g = g % 9`	Значение `3` присваивается переменной `g`

3.10. Операторы инкремента и декремента

В следующей таблице приведены описания двух унарных операторов языка C++. **Оператор инкремента (increment operator)** `++` прибавляет `1` к числовой переменной, а **оператор декремента (decrement operator)** `--` вычитает `1` из числовой переменной.

Оператор	Название оператора	Пример выражения	Пояснение
++	оператор префиксного инкремента	++number	Увеличить number на 1, затем использовать в выражении новое значение number
++	оператор постфиксного инкремента	number++	Использовать в выражении текущее значение number, затем увеличить number на 1
--	оператор префиксного декремента	--number	Уменьшить number на 1, затем использовать в выражении новое значение number
--	оператор постфиксного декремента	number--	Использовать в выражении текущее значение number, затем уменьшить number на 1

Операторы инкремента и декремента, которые ставятся перед переменной, называются **префиксными операторами (prefix operators)**. Операторы инкремента и декремента, которые ставятся после переменной, называются **постфиксными операторами (postfix operators)**.

Префиксный оператор инкремента (или декремента) прибавляет 1 к переменной (или вычитает из нее 1). После увеличения (или уменьшения) новое значение переменной используется в выражении с этим оператором.

Постфиксный оператор инкремента (или декремента) работает иначе. Он тоже прибавляет 1 к переменной (или вычитает из нее 1). Но в выражении с этим оператором используется не новое, а прежнее значение переменной. И только после этого значение переменной увеличивается (или уменьшается).

В отличие от бинарных операторов, унарные операторы инкремента и декремента надо ставить рядом с их операндами без пробелов.

Различие между префиксным и постфиксным операторами инкремента

Иллюстрация 3.4 демонстрирует различие между префиксным и постфиксным операторами инкремента (++). Префиксный и постфиксный операторы декремента (--) работают аналогично.

```
// fig03_04.cpp
// Различие между префиксным и постфиксным операторами инкремента
#include <iostream>
using namespace std;

int main() {
   // Применяем постфиксный оператор
   int c{5};
   cout << "c before postincrement: " << c << "\n"; // Выводим число 5
   cout << " postincrementing c: " << c++ << "\n"; // Выводим число 5
```

```
11    cout << " c after postincrement: " << c << "\n"; // Выводим число 6
12
13    cout << "\n"; // Добавляем пустую строку
14
15    // Применяем префиксный оператор
16    c = 5;
17    cout << " c before preincrement: " << c << "\n"; // Выводим число 5
18    cout << " preincrementing c: " << ++c << "\n"; // Выводим число 6
19    cout << " c after preincrement: " << c << "\n"; // Выводим число 6
20 }
```

```
c before postincrement: 5
   postincrementing c: 5
 c after postincrement: 6

 c before preincrement: 5
    preincrementing c: 6
  c after preincrement: 6
```

Илл. 3.4 | Различие между префиксным и постфиксным операторами инкремента

В строке 8 переменная `c` инициализируется значением `5`, которое выводится в строке 9. В строке 10 выводится результат выражения `c++`. Поскольку оператор инкремента постфиксный, в этом выражении используется (выводится на экран) прежнее значение `5` и только после этого переменная получает новое значение `6`. В строке 11 оно выводится на экран, и в строке 10 мы видим, что переменная изменилась.

В строке 16 переменной возвращается исходное значение `5`, которое выводится на экран в строке 17. В строке 18 выводится результат выражения `++c`. Поскольку оператор инкремента префиксный, в этом выражении используется (выводится на экран) уже новое значение `6`. В строке 19 оно снова выводится на экран, и мы видим, что после выполнения строки 18 переменная не изменилась.

Использование операторов для упрощения инструкций

Арифметические составные операторы присваивания, а также операторы инкремента и декремента позволят вам записывать многие инструкции ваших программ более компактно. Например, в коде, показанном на илл. 3.3, есть три инструкции присваивания в строках 21, 24 и 28:

```
passes = passes + 1;
failures = failures + 1;
studentCounter = studentCounter + 1;
```

С составным оператором присваивания эти же инструкции можно записать так:

```
passes += 1;
failures += 1;
studentCounter += 1;
```

Аналогичные операции можно выполнять с префиксным оператором инкремента:

```
++passes;
++failures;
++studentCounter;
```

и с постфиксным оператором инкремента:

```
passes++;
failures++;
studentCounter++;
```

Если вся инструкция заключается в увеличении или уменьшении значения переменной (без участия этой переменной в каком-либо выражении), то префиксный и постфиксный операторы приводят к одинаковому результату. И только если переменная используется в выражении, выбор между префиксным и постфиксным операторами инкремента (или декремента) будет влиять на результат.

Попытка использовать операторы инкремента и декремента в выражениях без присваивания приводит к ошибке компиляции. Например, инструкция `++(x + 1)` будет синтаксической ошибкой, поскольку `(x + 1)` не является переменной.

Приоритеты и правила группировки операторов

В следующей таблице показаны приоритеты и правила группировки тех операторов C++, о которых вы уже знаете. Все они перечислены сверху вниз в порядке убывания приоритетов. Обратите внимание, что тернарный условный оператор (`?:`), унарные префиксные операторы инкремента (`++`) и декремента (`--`), операторы сложения (`+`) и вычитания (`-`), а также операторы присваивания (`=`, `+=`, `-=`, `*=`, `/=` и `%=`) группируются **справа налево**. Все остальные операторы из этой таблицы группируются **слева направо**.

Операторы[1]						Группировка ассоциативность
`++`	`--`	`static_cast<`*тип*`>`			`()`	Слева направо
`++`	`--`	`+`	`-`			Справа налево
`*`	`/`	`%`				Слева направо
`+`	`-`					Слева направо
`<<`	`>>`					Слева направо
`<`	`<=`	`>`	`>=`			Слева направо
`==`	`!=`					Слева направо
`?:`						Справа налево
`=`	`+=`	`-=`	`*=`	`/=`	`%=`	Справа налево

[1] Обратите внимание, что операторы могут выглядеть совершенно одинаково, но при этом отличаться по своему действию, по приоритету и по ассоциативности. Приложение А в конце книги поможет вам уверенно различать такие операторы. — *Примеч. ред.*

3.11. Несовместимость основных типов данных на разных платформах

Здесь вы можете просмотреть полный список фундаментальных (основных) типов данных языка C++ и допустимые диапазоны значений для каждого типа:

https://en.cppreference.com/w/cpp/language/types

Как видите, эти диапазоны зависят от платформы (то есть от программно-аппаратной конструкции компьютера). В языках C и C++ число типа `int` на одной платформе может занимать в памяти 16 бит (2 байта) памяти, на другой — 32 бита (4 байта), а на третьей — 64 бита (8 байт). Поэтому код с переменными типа `int`, написанный для конкретной платформы, не обязательно будет работать на других платформах. Программист может написать несколько версий приложения для разных платформ или использовать методы, гарантирующие нужный уровень совместимости[1]. В следующем разделе вы узнаете об одном из этих методов.

Рассмотрим причину названной проблемы на примере целочисленных типов `int`, `long` и `long long`. Стандарт C++ требует, чтобы размер переменных `int` в памяти был не менее 16 бит, переменных `long` — не менее 32 бит, а переменных `long long` — не менее 64 бит. Кроме того, переменные `int` должны быть по размеру не больше, чем `long`, а переменные `long` — не больше, чем `long long`. Из-за таких мягких требований иногда возникают проблемы с совместимостью, зато разработчики могут добиваться максимального быстродействия программ на каждой из платформ, сопоставляя размеры основных типов данных с конструктивными особенностями компьютеров.

3.12. Готовые объекты: работа с целыми числами произвольной величины

Допустимый диапазон значений для конкретного целочисленного типа данных зависит от того, сколько байтов занимает каждая переменная этого типа в памяти компьютера той или иной конструкции. Например, если переменная типа `int` занимает `4` байта, она может хранить 2^{32} возможных значений, то есть любое целое число в диапазоне от `-2 147 483 648` до `2 147 483 647`. На большинстве современных платформ переменная типа `long long` занимает в памяти 8 байт, то есть может хранить 2^{64} возможных значений в диапазоне от `-9 223 372 036 854 775 808` до `9 223 372 036 854 775 807`.

В некоторых приложениях нужны числа за пределами диапазона long long

Рассмотрим вычисление факториала. Факториал — это произведение всех целых чисел от 1 до заданного числа. Факториал числа 5 (пишется как `5!`) равен

[1] Например, переменные целочисленных типов, определенных в заголовке <cstdint> (https://en.cppreference.com/w/cpp/types/integer), будут корректно работать в вашем приложении на разных платформах.

1 · 2 · 3 · 4 · 5, что составляет 120. Самое большое значение факториала, которое можно хранить в 64-битной переменной типа `long long`, — это 20!, что составляет `2 432 902 008 176 640 000`. Факториалы быстро выходят за пределы диапазона `long long`. И это лишь один характерный пример. В нашу эпоху для обработки быстро растущих объемов «больших данных» все больше реальных приложений должны выполнять такие вычисления, для которых типа данных `long long` уже недостаточно.

Одна из сфер применения чрезвычайно больших целых чисел — это криптография, необходимая для защиты информации, передаваемой между компьютерами через интернет. Многие криптографические алгоритмы выполняют вычисления со 128-битными или 256-битными целыми числами — значительно большими, чем можно хранить в переменных основных типов C++.

Класс BigNumber для работы с целыми числами произвольной величины

Любому приложению, использующему целые числа за пределами диапазона `long long`, нужен специально предназначенный для этого класс. К сожалению, в стандартной библиотеке C++ все еще нет класса для целых чисел произвольной величины. Поэтому мы погрузимся в огромный мир библиотек C++ с открытым исходным кодом, чтобы продемонстрировать вам один из тех классов, которые подходят для этой цели. Это класс `BigNumber`, опубликованный на портале GitHub:

https://github.com/limeoats/BigNumber

Для вашего удобства мы поместили этот класс в папку `libraries`, которая находится в папке `examples`. Ознакомьтесь с файлом `LICENSE.md`, содержащим условия лицензирования открытого исходного кода.

Для работы с классом `BigNumber` вам не обязательно понимать, как он реализован[1]. Вы просто включаете в вашу программу его заголовочный файл `bignumber.h`, легко создаете объекты этого класса, а затем используете их в своем коде. На илл. 3.5 вы видите код программы с классом `BigNumber`, а также пример вывода данных. Мы взяли самое длинное число типа `long long`, чтобы показать, как создавать еще большие числа с помощью `BigNumber`. В конце этого раздела мы покажем, как скомпилировать и запустить эту программу с помощью выбранного вами компилятора.

```
1 // fig03_05.cpp
2 // Работа с целыми числами произвольной величины
3 #include <iostream>
4 #include "bignumber.h"
5 using namespace std;
6
7 int main() {
8    // Работаем с самым большим числом типа long long
```

[1] После глубокого погружения в C++ вы, возможно, захотите взглянуть на исходный код класса BigNumber (примерно 1000 строк), чтобы понять, как он реализован. А в тех главах книги, которые посвящены объектно-ориентированному программированию, вы узнаете о нескольких способах написать ваш собственный класс, работающий с очень большими целыми числами.

```
    const long long value1{9'223'372'036'854'775'807LL}; // Предел long long
    cout << "long long value1: " << value1
       << "\nvalue1 - 1 = " << value1 - 1 // Правильный результат
       << "\nvalue1 + 1 = " << value1 + 1; // Неправильный результат

    // Работаем с числом типа BigNumber
    const BigNumber value2{value1};
    cout << "\n\nBigNumber value2: " << value2
       << "\nvalue2 - 1 = " << value2 - 1 // Правильный результат
       << "\nvalue2 + 1 = " << value2 + 1; // Правильный результат

    // Вычисляем степени числа 100 000 000 типа long long
    long long value3{100'000'000};
    cout << "\n\nvalue3: " << value3;

    int counter{2};

    while (counter <= 5) {
       value3 *= 100'000'000; // Скоро нам не хватит диапазона long long
       cout << "\nvalue3 to the power " << counter << ": " << value3;
    ++counter;
    }

    // Вычисляем степени числа 100 000 000 типа BigNumber
    BigNumber value4{100'000'000};
    cout << "\n\nvalue4: " << value4 << "\n";

    counter = 2;

    while (counter <= 5) {
       cout << "value4.pow(" << counter << "): "
          << value4.pow(counter) << "\n";
       ++counter;
    }

    cout << "\n";
}
```

```
long long value1: 9223372036854775807
value1 - 1: 9223372036854775806                    (OK)
value1 + 1: -9223372036854775808                   (Ошибка!)

BigNumber value2: 9223372036854775807
value2 - 1: 9223372036854775806                    (OK)
value2 + 1: 9223372036854775808                    (OK)

value3: 100000000
value3 to the power 2: 10000000000000000           (OK)
value3 to the power 3: 2003764205206896640         (Ошибка!)
value3 to the power 4: -8814407033341083648        (Ошибка!)
value3 to the power 5: -5047021154770878464        (Ошибка!)
```

Окончание ⇨

```
value4: 100000000
value4.pow(2): 10000000000000000                                    (OK)
value4.pow(3): 1000000000000000000000000                            (OK)
value4.pow(4): 100000000000000000000000000000000                    (OK)
value4.pow(5): 10000000000000000000000000000000000000000           (OK)
```

Илл. 3.5 | Работа с целыми числами произвольной величины

Включение заголовка, которого нет в стандартной библиотеке C++

В директиве `#include` имена заголовков, расположенных в папке с исходным кодом приложения (или в одной из вложенных в нее папок), заключаются в кавычки (""), а не в угловые скобки (<>). Кавычки сообщают компилятору, что заголовок надо взять из папки вашего приложения или из другой указанной вами папки.

Что случится, если превысить предельное значение long long?

В строке 9 мы инициализируем переменную `value1` максимально возможным для нашей платформы[1] значением типа `long long`:

```
const long long value1{9'223'372'036'854'775'807LL}; // Предел long long
```

При вводе с клавиатуры больших чисел (длинных последовательностей цифр) не исключены ошибки. Чтобы снизить риск ошибок и сделать такие числа более простыми для восприятия, в C++14 был введен **разделитель цифр (digit separator)** ' — символ одиночной кавычки, который вставляется между разрядами. Мы разделили длинное число на разряды из трех цифр. Обратите внимание на символы `LL` в конце числа — они означают, что это число типа `long long`.

В строке 10 мы выводим значение переменной `value1`, а в строке 11 вычитаем из него `1`, чтобы показать, что эта операция будет выполнена правильно. Затем мы пробуем прибавить `1` к `value1`, хотя эта переменная уже содержит максимальное значение `long long`. Все наши компиляторы не справились с этим и ошибочно вывели на экран минимальное значение `long long`. Согласно стандарту C++, вычисление с выходом за пределы допустимого диапазона приводит к **неопределенному поведению (undefined behavior)** программы. Проявление этой неопределенности зависит от платформы. Например, мы получили неправильный результат, а на какой-то другой платформе программа завершилась бы с сообщением об ошибке. Это еще один пример несовместимости основных целочисленных типов в версиях языка C++ для разных платформ.

---

[1] Мы протестировали этот код на нескольких платформах и выяснили, что на каждой из них числа long long не могут быть больше чем 9 223 372 036 854 775 807. Кстати, этот предел можно найти программно, с помощью выражения std::numeric_limits<long long>::max(), использующего класс numeric_limits из заголовка стандартной библиотеки C++ <limits>. Но обозначения <> и :: будут рассмотрены в последующих главах, а сейчас мы прямо указали в коде предельно возможное число: 9 223 372 036 854 775 807.

Выполнение тех же операций с объектом BigNumber

В строках 15–18 мы повторяем операции из строк 9–12, но теперь уже с числами типа `BigNumber`. Мы создаем объект `BigNumber` с именем `value2` и инициализируем его значением переменной `value1` — тем самым, которое невозможно превзойти в переменных типа `long long`:

```
BigNumber value2{value1};
```

Далее мы выводим значение `BigNumber`, затем вычитаем из него `1` и выводим результат. В строке 18 мы прибавляем `1` к `value2` (то есть к максимально возможному числу `long long`). Числа типа `BigNumber` могут быть любой величины, поэтому теперь вычисление выполняется правильно. Мы получили результат, недостижимый при работе с основными целочисленными типами C++ (по крайней мере, на имеющихся у нас компьютерах).

Класс `BigNumber` поддерживает все стандартные арифметические операторы, в том числе операторы сложения (`+`) и вычитания (`-`), которые есть в этой программе. Компилятор C++ по умолчанию умеет применять арифметические операторы к основным числовым типам C++, но ему надо «объяснить», как применять эти же операторы к объектам сторонних классов. Мы обсудим этот процесс, называемый перегрузкой операторов, в главе 11.

Возведение в степень числа 100 000 000 с целыми числами long long

В строках 21–30 число `100 000 000` типа `long long` возводится во 2-ю, 3-ю, 4-ю и 5-ю степень. Сначала мы создаем переменную `value3` и отображаем ее значение. Строки 26–30 повторяются в цикле 4 раза. Инструкция:

```
value3 *= 100'000'000; // Скоро нам не хватит диапазона long long
```

умножает текущее значение `value3` на `100 000 000`. Это значит, что каждый раз `value3` возводится в следующую степень. Выходные данные показывают, что вычисление выполняется правильно только в первый раз.

Возведение в степень числа 100 000 000 с объектом BigNumber

Теперь мы демонстрируем возможность класса `BigNumber` работать с числами, значительно превосходящими пределы диапазонов всех основных целочисленных типов. В строках 33–42 число `100 000 000` снова возводится во 2-ю, 3-ю, 4-ю и 5-ю степень, но теперь оно имеет тип `BigNumber`. Сначала мы создаем переменную `value4` и выводим ее значение. Строки 38–42 повторяются в цикле 4 раза. Этот фрагмент кода вызывает функцию `pow` класса `BigNumber`, чтобы возвести значение переменной `value4` в степень `counter`[1]:

```
value4.pow(counter)
```

[1] Мы решили показать вам функцию pow из класса BigNumber, но могли бы использовать и оператор *=, как в строке 27.

Благодаря классу `BigNumber`, все вычисления выполняются корректно. Мы получаем огромные числа, что было бы невозможно с основными целочисленными типами на платформах Windows, macOS и Linux.

Класс `BigNumber` позволяет работать с любыми целыми числами независимо от вашей платформы, то есть на совершенно разных компьютерах результаты будут одинаковыми. Но вычисления выполняются медленнее, чем с основными целочисленными типами. Таким образом, вы жертвуете быстродействием в обмен на универсальность, предоставляемую классом `BigNumber`.

Компиляция и запуск программы с помощью Microsoft Visual Studio

В интегрированной среде разработки Microsoft Visual Studio:

1. Создайте новый проект, как описано в разделе 1.2.
2. В `Solution Explorer` щелкните правой кнопкой мыши на `Source Files` проекта и выберите `Add > Existing Item...`.
3. Перейдите в папку `fig03_05`, выберите `fig03_05.cpp` и нажмите кнопку `Add`.
4. Повторите шаги 2 и 3 для файла `bignumber.cpp` из `examples\libraries\BigNumber\src`.
5. В `Solution Explorer` щелкните правой кнопкой мыши на имени проекта и выберите `Properties...`.
6. В разделе `Configuration Properties` выберите `C/C++`. Затем в правой части диалогового окна добавьте в `Additional Include Directories` полный путь к папке `BigNumber\src` на своем компьютере. На нашем компьютере этот путь выглядит так:

   `C:\Users\`*`ИмяПользователя`*`\Documents\examples\libraries\BigNumber\src`

7. Нажмите `OK`.
8. Нажмите `Ctrl` + `F5` для компиляции и запуска программы[1].

Компиляция и запуск программы с помощью GNU g++

Если вы работаете с компилятором g++ из пакета GNU Compiler Collection (или в окне терминала macOS):

1. В командной строке перейдите в папку `fig03_05`, содержащую исходный код программы.
2. Чтобы скомпилировать программу, введите (в одну строку) следующую команду (параметр `-I` указывает на дополнительные папки, в которых компилятор будет искать заголовочные файлы):

```
g++ -std=c++2a -I ../../libraries/BigNumber/src fig03_05.cpp
   ../../libraries/BigNumber/src/bignumber.cpp -o fig03_05
```

[1] Visual C++ может выдавать предупреждения, если в коде есть класс BigNumber.

3. Введите следующую команду для запуска программы:

```
./fig03_05
```

Компиляция и запуск программы с помощью Apple Xcode

В интегрированной среде разработки Apple Xcode:

1. Создайте новый проект, как описано в разделе 1.2, и удалите `main.cpp`.
2. Перетащите `fig03_05.cpp` из папки `fig03_05` в папку с исходным кодом вашего проекта в Xcode, затем нажмите кнопку `Finish` в появившемся диалоговом окне.
3. Перетащите `bignumber.h` и `bignumber.cpp` из папки `examples/libraries/BigNumber/src` в папку с исходным кодом вашего проекта в Xcode, затем нажмите кнопку `Finish` в появившемся диалоговом окне.
4. Нажмите `Command + R` для компиляции и запуска программы.

3.13. Форматирование текста функцией format

В C++20 появились новые возможности форматирования текстовых строк с помощью функции `format` (для работы с ней достаточно включить в программу заголовок `<format>`). Форматирование стало значительно проще благодаря новому синтаксису — примерно такому же, как в языке Python, в языках Microsoft.NET (C# и Visual Basic) и в новом языке Rust[1]. По сравнению с более ранними версиями C++ новый синтаксис стал лаконичнее и мощнее. Далее мы будем в первую очередь использовать эти новые возможности.

Форматирование строк по стандарту C++20 еще не реализовано в некоторых компиляторах

На момент написания этой главы (декабрь 2021 года) только Visual C++ поддерживает все возможности форматирования текста, предусмотренные стандартом C++20. Другие компиляторы пока опаздывают, но такие же возможности дает библиотека с открытым исходным кодом `{fmt}`, опубликованная на портале GitHub[2,3]:

https://github.com/fmtlib/fmt

---

[1] Виктор Зверович (Victor Zverovich), «Text Formatting», 16 июля 2019 г. Был доступен: 15 сентября 2023 г. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0645r10.html.

[2] Согласно предложению Комитета по стандартам C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0645r10.html.

[3] Средства форматирования текста по стандарту C++20 — это лишь часть функциональности библиотеки {fmt}.

В следующий пример программы мы включили библиотеку {fmt}[1]. Для вашего удобства мы поместили ее в папку `libraries`, которая находится в папке `examples`. В конце этого раздела мы покажем, как скомпилировать и запустить эту программу с помощью выбранного вами компилятора. Ознакомьтесь с условиями лицензирования библиотеки по адресу:

https://github.com/fmtlib/fmt/blob/master/LICENSE.rst

Строка формата с заполнителями

Первым аргументом функции `format` является **строка формата (format string)**, содержащая **заполнители (placeholders)**, которые определяются фигурными скобками ({ и }). Функция `format` заменяет эти заполнители значениями других своих аргументов, как показано на илл. 3.6.

```
// fig03_06.cpp
// Форматирование строк в C++20
#include <iostream>
#include <fmt/format.h> // Или в C++20: #include <format>
using namespace std;
using namespace fmt; // Не требуется в C++20

int main() {
   string student{"Paul"};
   int grade{87};

   cout << format("{}'s grade is {}\n", student, grade);
}
```

```
Paul's grade is 87
```

Илл. 3.6 | Форматирование строк в C++20

Заполнители заменяются слева направо

По умолчанию функция `format` заменяет заполнители в строке формата слева направо. Мы вызываем функцию `format`, передавая ей три аргумента. Первый из них — следующая строка формата:

```
"{}'s grade is {}\n"
```

[1] Некоторые наши примеры кода, связанные с нововведениями в C++20, могут быть несовместимы с вашим компилятором. Когда стандарт C++20 получит полную поддержку во всех основных компиляторах, мы повторно протестируем примеры кода, обновим их и опубликуем обновления для книги на нашем сайте: https://deitel.com/c-plus-plus-20-for-programmers. Пример кода в этом разделе компилируется, но требует библиотеку с открытым исходным кодом {fmt} для демонстрации тех возможностей, которые со временем будут поддерживаться всеми компиляторами C++20.

Функция заменяет в этой строке первый заполнитель значением переменной `student` (`Paul`), а второй заполнитель — значением переменной `grade` (`87`), после чего возвращает готовую строку:

```
"Paul's grade is 87"
```

Компиляция и запуск программы с помощью Microsoft Visual Studio

Выполните действия, аналогичные тем, которые описаны в разделе 3.12, но со следующими изменениями:

- На шаге 3 перейдите в папку `fig03_06` и добавьте `fig03_06.cpp` в `Source Files` вашего проекта.
- На шаге 4 перейдите в папку `examples\libraries\fmt\src` и добавьте файл `format.cc` в `Source Files` вашего проекта.
- На шаге 6 добавьте в `Additional Include Directories` полный путь к папке `include` библиотеки `{fmt}` на своем компьютере. На нашем компьютере этот путь выглядит так:

  `C:\Users\`*`ИмяПользователя`*`\Documents\examples\libraries\fmt\include`

Компиляция и запуск программы с помощью GNU g++

Если вы работаете с компилятором g++ из пакета GNU Compiler Collection (или в окне терминала macOS):

1. В командной строке перейдите в папку `fig03_06`, содержащую исходный код программы.
2. Чтобы скомпилировать программу, введите (в одну строку) следующую команду:

   ```
   g++ -std=c++2a -I ../../libraries/fmt/include fig03_06.cpp
      ../../libraries/fmt/src/format.cc -o fig03_06
   ```

3. Введите следующую команду для запуска программы:

   ```
   ./fig03_06
   ```

Компиляция и запуск программы с помощью Apple Xcode

Выполните действия, аналогичные тем, которые описаны в разделе 3.12, но с одним изменением: на шаге 2 перетащите файлы `fig03_06.cpp` и `format.cc`, а также папку `fmt` из папки `fig03_06` в папку с исходным кодом вашего проекта в Xcode.

3.14. Итоги

Для разработки любого алгоритма нужны только три вида управляющих инструкций: последовательность, выбор (ветвление) и цикл. Мы познакомили вас

с инструкцией одиночного выбора `if`, с инструкцией двойного выбора `if...else` и с инструкцией цикла `while`. Мы использовали вложенные управляющие инструкции, чтобы суммировать оценки учащихся и вычислять среднюю оценку с помощью циклов двух видов: со счетчиком и с контрольным значением. Мы использовали вложенные управляющие инструкции для анализа результатов экзамена и принятия решений. Мы рассказали о составных операторах присваивания, а также об операторах инкремента и декремента.

Мы объяснили, почему основные типы данных C++ на разных платформах могут отличаться по размеру в памяти компьютера, а затем использовали объекты класса `BigNumber` с открытым исходным кодом, чтобы работать с числами, выходящими за пределы диапазона основных типов. Наконец, мы показали, как форматировать текст с помощью библиотеки `{fmt}` с открытым исходным кодом. Когда в будущей версии вашего компилятора появится полная поддержка заголовка `<format>` из C++20[1], вы сможете точно так же форматировать текст без библиотеки `{fmt}`. В главе 4 мы расскажем о других управляющих инструкциях: `for`, `do...while` и `switch`, а также рассмотрим формирование составных условий с помощью логических операторов.

[1] Как правило, за то время, пока готовятся к печати русские версии книг, первоначально изданных на английском языке, разработчики компиляторов успевают реализовать все новшества, упомянутые в этих книгах. Скорее всего, в тот момент, когда вы читаете эту книгу, ваш компилятор уже на 100% поддерживает весь приведенный в ней исходный код во всех вариантах. – *Примеч. ред.*



Управляющие инструкции (часть 2)

В этой главе вы:

- научитесь использовать инструкции циклов `for` и `do...while`;
- научитесь использовать инструкцию множественного выбора `switch`;
- узнаете, зачем нужен атрибут `[[fallthrough]]` в инструкции `switch`;
- научитесь использовать инструкции выбора с инициализаторами;
- научитесь использовать инструкции `break` и `continue`, чтобы изменять поток управления;
- научитесь использовать логические операторы для формирования составных условий;
- узнаете об ошибках представления, возникающих при денежных вычислениях с плавающей точкой;
- продолжите работать с готовыми объектами и узнаете, как создавать и распаковывать архивы в формате ZIP с помощью общедоступной библиотеки;
- продолжите изучение новых средств форматирования текста, введенных в C++20.

4.1. Введение

В этой главе мы расскажем об управляющих инструкциях `for`, `do...while`, `switch`, `break` и `continue`. Мы детально рассмотрим, как работают циклы со счетчиком. Мы покажем пример расчета сложных процентов, начнем исследование проблем обработки денежных величин и обсудим ошибки представления при вычислениях с плавающей точкой. Затем с помощью инструкции `switch` мы подсчитаем количество оценок A, B, C, D и F в наборе данных о результатах экзамена. Мы покажем, как появившиеся в C++17 усовершенствования синтаксиса позволяют вам инициализировать переменную (или несколько переменных одного типа) в заголовках инструкций `if` и `switch`. Мы расскажем о логических операторах, позволяющих формировать составные условия из простых. Продолжая тему готовых объектов, мы займемся созданием и распаковкой ZIP-архивов с помощью библиотеки `miniz-cpp` с открытым исходным кодом. В конце главы мы представим вам еще несколько мощных и выразительных средств форматирования текста из C++20.

4.2. Как работает цикл со счетчиком

В главе 3 вы познакомились с инструкцией цикла `while`. Теперь пришло время подробнее изучить работу цикла со счетчиком, чтобы формализовать следующие элементы:

- **управляющая переменная (control variable)**, также называемая **счетчиком (counter)** цикла;
- **начальное значение (initial value)** управляющей переменной;
- **инкремент (increment)** управляющей переменной, выполняемый в каждой итерации цикла;
- **условие продолжения цикла (loop-continuation condition)**, определяющее, надо ли продолжать цикл.

Рассмотрим следующий код, в котором числа от `1` до `10` выводятся в цикле `while` (илл. 4.1).

Все элементы цикла `while` определены в строках 7, 9 и 11. В строке 7 объявляется управляющая переменная `counter` целочисленного типа `int`, для нее резервируется место в памяти и выполняется инициализация значением `1`. Все **объявления (declarations)**, требующие инициализации, являются исполняемыми инструкциями. Объявления переменных, резервирующие место в памяти, называются **определениями (definitions)**, но мы обычно будем придерживаться общего термина «объявление», кроме тех случаев, когда это различие важно.

В строке 10 выводится на экран значение управляющей переменной `counter` — однократно в каждой итерации цикла. В строке 11 значение управляющей переменной увеличивается на `1` — тоже однократно в каждой итерации цикла. В строке 9 проверяется, является ли значение `counter` меньшим или равным `10` (это последнее

из тех чисел, для которых условие продолжения цикла остается истинным). Когда значение `counter` станет больше `10`, цикл завершится.

```
1 // fig04_01.cpp
2 // Цикл while со счетчиком (с управляющей переменной)
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7    int counter{1}; // Объявляем и инициализируем управляющую переменную
8
9    while (counter <= 10) { // Условие продолжения цикла
10      cout << counter << " ";
11   ++counter; // Инкремент управляющей переменной
12   }
13
14   cout << "\n";
15 }
```

```
1 2 3 4 5 6 7 8 9 10
```

Илл. 4.1 | Цикл while со счетчиком (с управляющей переменной)

Тип управляющей переменной в любом цикле обязательно должен быть целочисленным. Никакие типы данных с плавающей точкой для этого не годятся, поскольку их значения обычно являются приблизительными. При инкременте такой переменной могут возникать неточности, приводящие к неправильной проверке условия продолжения цикла и препятствующие его нормальному завершению.

4.3. Инструкция цикла for

Все элементы цикла `for` со счетчиком определяются в одной строке исходного кода. На илл. 4.2 показан вариант предыдущей программы, в котором мы заменили цикл `while` циклом `for`.

При выполнении инструкции `for` (занимающей строки 9–11) прежде всего объявляется и инициализируется значением `1` управляющая переменная `counter`. Далее программа проверяет условие продолжения цикла, указанное после первой точки с запятой (`counter <= 10`). Начальное значение `counter` равно `1`, поэтому условие истинно и это значение выводится на экран в строке 10. После выполнения инструкции в строке 10 выражение `++counter`, указанное после второй точки с запятой, увеличивает `counter` на `1`. Затем программа проверяет условие продолжения цикла, чтобы определить, надо ли переходить к следующей итерации. На данный момент значение `counter` равно `2`, то есть условие по-прежнему истинно. Поэтому программа снова выполняет инструкцию в строке 10. Этот процесс продолжается до тех пор, пока цикл не выведет на экран все числа от `1` до `10`. После этого `counter` примет значение `11`, то есть условие продолжения цикла станет ложным и цикл

завершится. Выполнение программы продолжится со следующей инструкции после цикла, то есть со строки 13.

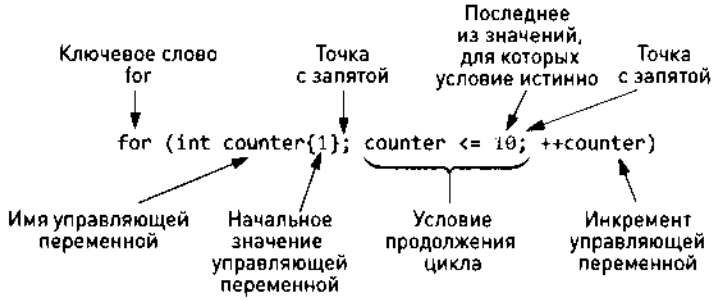
```
1 // fig04_02.cpp
2 // Цикл for со счетчиком (с управляющей переменной)
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7    // Заголовок цикла for содержит инициализацию,
8    // условие продолжения цикла и инкремент
9    for (int counter{1}; counter <= 10; ++counter) {
10      cout << counter << " ";
11   }
12
13   cout << "\n";
14 }
```

```
1 2 3 4 5 6 7 8 9 10
```

Илл. 4.2 | Цикл for со счетчиком (с управляющей переменной)

Заголовок инструкции for

На следующей диаграмме мы обозначили основные элементы инструкции `for` на примере строки 9 из илл. 4.2:



Эта первая часть инструкции (ключевое слово `for` и все то, что находится далее в круглых скобках) называется **заголовком (header)** инструкции `for`, или заголовком цикла `for`. Все элементы, от которых зависит работа такого цикла, сосредоточены в его заголовке.

Общий формат инструкции for

Общий формат инструкции `for` таков:

```
for (инициализация; условие продолжения цикла; инкремент) {
   тело цикла
}
```

В заголовок входят три секции, разделенные точкой с запятой:

- *Инициализация* — выражение, в котором определяется имя управляющей переменной и ей присваивается начальное значение.
- *Условие продолжения цикла* — выражение, при истинности которого выполняется тело цикла.
- *Инкремент* — выражение, изменяющее значение управляющей переменной таким образом, чтобы условие продолжения цикла стало ложным после необходимого количества повторений.

Если условие продолжения цикла изначально является ложным, то тело цикла `for` (одна или несколько инструкций между фигурными скобками) не будет выполнено ни разу, а выполнение программы продолжится со следующей инструкции после `for`.

Область видимости управляющей переменной

Управляющую переменную, объявленную в выражении инициализации инструкции `for`, можно использовать только в этой же инструкции, но не за ее пределами. Такое ограничение называется **областью видимости (scope)** переменной. От области видимости зависит, из каких частей программы можно обращаться к переменной и когда эта переменная перестанет существовать. Например, область видимости переменной начинается в точке ее объявления и заканчивается правой фигурной скобкой, закрывающей блок. Как вы увидите в главе 5, каждую переменную рекомендуется определять только в минимально необходимой для нее области видимости.

Выражения в заголовке инструкции for необязательны

Все три выражения в заголовке `for` являются необязательными. Если вы опустите *условие продолжения цикла*, оно всегда будет считаться истинным (автоматически), то есть вы получите бесконечный цикл. Вы можете опустить выражение *инициализации*, если управляющая переменная инициализирована в программе еще до начала цикла. Вы можете опустить выражение *инкремента*, если в теле цикла есть инструкции, изменяющие управляющую переменную.

Указанное в заголовке `for` выражение инкремента фактически выполняется программой как отдельная инструкция в конце тела цикла. Поэтому выражение инкремента в заголовке можно заменить любым из следующих эквивалентных выражений в теле цикла:

```
counter = counter + 1
counter += 1
++counter
counter++
```

В этом случае выражение инкремента не является частью какого-либо другого выражения, поэтому префиксный и постфиксный операторы инкремента работают

одинаково. Мы предпочитаем оператор префиксного инкремента: он дает небольшое преимущество в быстродействии, как вы увидите в главе 11, посвященной перегрузке операторов.

Управляющая переменная в теле цикла

Многие программы выводят на экран меняющееся значение управляющей переменной или используют его в вычислениях в теле цикла, но это не обязательно делать. Значение управляющей переменной можно менять из тела цикла, но это может привести к трудно выявляемым ошибкам. Если программа должна менять значение управляющей переменной в теле цикла, лучше использовать цикл `while` вместо `for`.

UML-диаграмма цикла for

Ниже приведена диаграмма действий цикла `for` из кода, показанного на илл. 4.2. Диаграмма показывает, что инициализация управляющей переменной происходит только один раз: перед первой проверкой условия. Инкремент этой переменной происходит после каждого выполнения инструкции в теле цикла:



4.4. Инкремент и декремент счетчика в цикле for

В следующих примерах показаны варианты изменения значений управляющей переменной в цикле `for`. В каждом примере мы показываем только заголовок. Обратите внимание, что значения управляющей переменной можно не только увеличивать, но и уменьшать соответствующими операторами (декремент вместо инкремента).

А. Увеличение от **1** до **100** с шагом **1**:

```
for (int i{1}; i <= 100; ++i)
```

Б. Уменьшение со **100** до **1** с шагом **1**:

```
for (int i{100}; i >= 1; --i)
```

В. Увеличение от 7 до 77 с шагом 7:

```
for (int i{7}; i <= 77; i += 7)
```

Г. Уменьшение с 20 до 2 с шагом 2:

```
for (int i{20}; i >= 2; i -= 2)
```

Д. Возрастающий ряд чисел 2, 5, 8, 11, 14, 17, 20:

```
for (int i{2}; i <= 20; i += 3)
```

Е. Убывающий ряд чисел 99, 88, 77, 66, 55, 44, 33, 22, 11, 0:

```
for (int i{99}; i >= 0; i -= 11)
```

Не используйте операторы равенства (!= или ==) в условии цикла, если управляющая переменная увеличивается или уменьшается более чем на 1. Например, посмотрите на ошибочно составленный цикл:

```
for (int counter{1}; counter != 10; counter += 2)
```

Здесь условие `counter != 10` никогда не станет ложным (и цикл будет повторяться бесконечно), потому что увеличение `counter` на 2 после каждого выполнения тела цикла дает в результате только нечетные числа (3, 5, 7, 9, 11, ...).

4.5. Программа для вычисления суммы четных чисел

Программа, показанная на илл. 4.3, использует цикл `for` для вычисления суммы четных целых чисел в диапазоне от 2 до 20 и сохраняет результат в переменной `total`. В каждой итерации цикла (строки 10–12) значение управляющей переменной `number` прибавляется к значению переменной `total`.

```
 1 // fig04_03.cpp
 2 // Суммирование целых чисел в цикле for
 3 #include <iostream>
 4 using namespace std;
 5
 6 int main() {
 7    int total{0};
 8
 9    // Суммируем все четные числа от 2 до 20
10    for (int number{2}; number <= 20; number += 2) {
11       total += number;
12    }
13
14    cout << "Sum is " << total << "\n";
15 }
```

```
Sum is 110
```

Илл. 4.3 | Суммирование целых чисел в цикле for

Выражения инициализации и инкремента в заголовке цикла `for` могут состоять из нескольких выражений, разделенных запятыми. Хотя это не рекомендуется, вы можете перенести инструкцию из тела цикла `for` (строка 11) в секцию инкремента, используя запятую:

```
for (int number{2}; number <= 20; total += number, number += 2) { }
```

Запятая между выражениями `total += number` и `number += 2` — это **оператор «запятая» (comma operator)**. Он гарантирует, что выражения в списке будут вычисляться слева направо. Запятая имеет наименьший приоритет из всех операторов C++. Значением и типом всего списка выражений, разделенных запятыми, считаются значение и тип крайнего правого выражения соответственно. Оператор «запятая» часто используется в таких инструкциях `for`, где нужны несколько инициализирующих выражений или несколько выражений, меняющих значения управляющей переменной.

4.6. Программа для расчета сложных процентов

В этом разделе мы выполним расчет сложных финансовых процентов с помощью цикла `for`. Рассмотрим следующую задачу:

> *Человек делает вклад размером 1000 долларов на сберегательный счет с доходностью 5% годовых. Предполагая, что все проценты зачисляются на этот же счет, надо рассчитать и вывести на экран сумму на счете в конце каждого года на протяжении 10 лет. Используйте следующую формулу:*
>
> $a = p\,(1 + r)^n$
>
> *где:*
>
> *p — первоначальный вклад (то есть основная сумма);*
>
> *r — годовая процентная ставка (5% — это 0.05);*
>
> *n — количество лет;*
>
> *a — сумма на счете в конце года с номером n.*

Решение показано на илл. 4.4. Для расчета результата каждого года в течение 10 лет используется цикл. Мы выбрали для денежных вычислений тип данных с плавающей точкой: `double`. Вскоре мы обсудим проблемы, возникающие при денежных вычислениях с плавающей точкой. Для финансовых приложений, требующих абсолютно точных денежных вычислений с контролем округления, можно использовать, например, библиотеку с открытым исходным кодом `Boost.Multiprecision`[1].

---

[1] Джон Мэддок (John Maddock) и Кристофер Корманьос (Christopher Kormanyos), «Chapter 1. Boost.Multiprecision». Был доступен: 15 сентября 2023 г. https://www.boost.org/doc/libs/master/libs/multiprecision/doc/html/index.html.

```
1 // fig04_04.cpp
2 // Расчет сложных процентов в цикле for
3 #include <iostream>
4 #include <iomanip>
5 #include <cmath> // Для функции возведения в степень
6 using namespace std;
7
8 int main() {
9    // Устанавливаем формат чисел с плавающей точкой
10   cout << fixed << setprecision(2);
11
12   double principal{1000.00}; // Первоначальный вклад
13   double rate{0.05}; // Процентная ставка
14
15   cout << "Initial principal: " << principal << "\n";
16   cout << " Interest rate: " << rate << "\n";
17
18   // Выводим заголовки таблицы
19   cout << "\nYear" << setw(20) << "Amount on deposit" << "\n";
20
21   // Рассчитываем сумму на счете в конце каждого года из десяти
22   for (int year{1}; year <= 10; ++year) {
23      // Рассчитываем сумму на счете в конце текущего года
24      double amount{principal * pow(1.0 + rate, year)} ;
25
26      // Выводим на экран год и сумму
27      cout << setw(4) << year << setw(20) << amount << "\n";
28   }
29 }
```

```
Initial principal: 1000.00
Interest rate: 0.05

Year   Amount on deposit
   1             1050.00
   2             1102.50
   3             1157.63
   4             1215.51
   5             1276.28
   6             1340.10
   7             1407.10
   8             1477.46
   9             1551.33
  10             1628.89
```

Илл. 4.4 | Расчет сложных процентов в цикле for

В строках 12–13 инициализируются две переменные типа `double`: `principal` получает значение `1000.00`, а `rate` — `0.05`. Компилятор C++ воспринимает числа с десятичной точкой (такие, как `1000.00` и `0.05`) как числа типа `double`, а целые

числа (такие, как `7` и `-22`) — как числа типа `int`[1]. В строках 15–16 выводится на экран первоначальная величина вклада и процентная ставка.

Форматирование текста с выравниванием по ширине

Сочетание строки 10 перед циклом и строки 27 в теле цикла выводит значения переменных `year` и `amount`. Мы задаем формат вывода с помощью параметризованных манипуляторов потока `setprecision` и `setw` и непараметризованного манипулятора потока `fixed`. Манипулятор потока `setw(4)` определяет **ширину поля (field width)** для следующего выводимого на экран значения: 4 символа. Если выводимое значение содержит не более 4 символов, то есть помещается в указанную ширину поля (как в нашем примере с номером года), оно по умолчанию выравнивается по правому краю поля. Если выводимое значение не помещается в эту ширину, компилятор C++ все равно выводит его полностью, для чего расширяет поле вправо. Для выравнивания по левому краю поля добавьте непараметризованный манипулятор потока `left` (заголовок `<iostream>`). Чтобы вернуть выравнивание по правому краю, добавьте непараметризованный манипулятор потока `right`.

Другие средства форматирования в инструкциях вывода отображают переменную `amount` в виде числа с фиксированной десятичной точкой (`fixed` в строке 10) и с двумя цифрами после точки (`setprecision(2)` в строке 10), которое выровнено по правому краю в поле шириной `20` символов (`setw(20)` в строке 27). Напомним, что манипуляторы потока `fixed` и `setprecision` относятся к липким настройкам, то есть действуют до тех пор, пока вы их явно не измените. Мы применили их к потоку `cout` до начала цикла `for`, поэтому их уже не надо применять внутри цикла. Но ширина поля, указанная с помощью `setw`, не является липкой настройкой и действует только для текущего выводимого значения. В разделе 4.14 вы узнаете о других возможностях форматирования текста, добавленных в C++20.

Расчет процентов с помощью функции pow стандартной библиотеки

Цикл `for` (строки 22–28) повторяется `10` раз, при этом значение управляющей переменной `year` возрастает от `1` до `10` с шагом `1`. Переменная `year` представляет число n в формуле из постановки задачи.

В C++ нет оператора возведения в степень, поэтому нам понадобилась функция `pow` стандартной библиотеки C++ (строка 24) из заголовка `<cmath>` (строка 5). Вызов функции `pow(x, y)` возвращает результат возведения числа `x` в степень `y`. Функция получает два аргумента типа `double` и возвращает значение этого же типа. Строка 24 выполняет вычисление по формуле $a = p(1 + r)^n$, где a — `amount`, p — `principal`, r — `rate` и n — `year`.

Первый аргумент функции `pow` — это выражение `1.0 + rate`. Оно не меняется в итерациях цикла, поэтому нет смысла вычислять его в теле цикла десять раз подряд.

---

[1] В разделе 3.12 мы показали, что целочисленных типов данных C++ недостаточно, чтобы хранить целые числа любой величины. Всегда выбирайте тип данных, подходящий для того диапазона значений, который вам нужен. Если вы хотите указать, что целочисленный литерал имеет тип long или long long, добавьте после литерала символы L или LL соответственно.

Многие современные оптимизирующие компиляторы в таких случаях выполняют вычисление однократно, до начала цикла, используя в скомпилированном коде цикла готовый результат. Это ускоряет работу программ.

Точность чисел с плавающей точкой и требования к памяти

Тип данных `float` представляет **число с плавающей точкой одинарной точности (single-precision floating-point number)**. В большинстве современных компьютеров такое число занимает 4 байта памяти, что соответствует точности примерно в 7 значащих цифр. Тип данных `double` представляет **число с плавающей точкой двойной точности (double-precision floating-point number)**. В большинстве современных компьютеров такое число занимает 8 байт памяти, что соответствует точности примерно в 15 значащих цифр — значительно больше по сравнению с `float`. Для вычислений с плавающей точкой программисты обычно предпочитают тип `double`. Все встречающиеся в исходном коде числа с десятичной точкой (например, `3.14159`) компилятор C++ по умолчанию считает числами типа `double`. Такие числа в исходном коде называются **литералами с плавающей точкой (floating-point literals)**.

Согласно стандарту C++, тип `double` должен обеспечивать не меньшую точность, чем `float`. Кроме того, в C++ есть тип данных `long double`, который должен обеспечивать не меньшую точность, чем `double`. Напомним, что полный список основных (фундаментальных) типов данных языка C++ и их диапазонов опубликован здесь:

https://en.cppreference.com/w/cpp/language/types

Числа с плавающей точкой приблизительны

В обычной арифметике числа с плавающей точкой часто возникают в результате деления. Разделив `10` на `3`, вы получите `3.3333333...`, причем последовательность цифр `3` после десятичной точки повторяется бесконечно. Но для хранения любого подобного числа в переменной компьютер выделяет фиксированный объем памяти, поэтому значение такой переменной практически всегда будет приблизительным. Иными словами, типы данных с плавающей точкой (такие, как `double`) страдают от так называемой **ошибки представления (representational error)**. Если программист предполагает, что числа с плавающей точкой идеально точны (например, использует их в операциях сравнения, проверяя на равенство), это может привести к ошибкам в программе.

У типов данных с плавающей точкой есть множество применений, особенно для работы с измеряемыми величинами. Например, измеряемой величиной является температура человеческого тела (в норме — `36.6` градуса по Цельсию). Когда термометр показывает `36.6`, на самом деле это может быть `36.659473213...`, но нам не нужны все эти цифры. Представление температуры тела с тремя значащими цифрами (например, `36.6`) хорошо подходит для реальных приложений медицинского характера, то есть в данном случае вполне достаточно типа `float`. И все-таки в большинстве других приложений тип данных `double` предпочтительнее, поскольку он представляет числа с плавающей точкой намного точнее, чем `float`[1].

[1] Современный стандарт представления чисел с плавающей точкой — IEEE 754 (https://en.wikipedia.org/wiki/IEEE_754).

Неточности в отображении округленных чисел

В примере с расчетом сложных процентов мы выбрали для переменных `amount`, `principal` и `rate` тип `double`. К сожалению, типы с плавающей точкой могут приводить к проблемам при денежных вычислениях. Посмотрите, что может случиться, если вы возьмете две переменные с плавающей точкой для представления денежных величин с двумя цифрами после точки (доллары и центы). Предположим, первая переменная хранит число `14.234` (округляемое до `14.23` при выводе на экран), а вторая — число `18.673` (округляемое до `18.67` при выводе на экран). При сложении этих чисел компилятор получит число `32.907`, которое будет представлено на экране как `32.91`. И ваша программа выведет такие данные:

```
  14.23
+ 18.67
  32.91
```

Человек, суммирующий числа в том виде, в каком они отображаются на экране, воспримет это как ошибку программы, так как ожидает увидеть сумму `32.90`. Имейте это в виду!

Пример ошибки представления денежных величин в числах с плавающей точкой

Ошибки представления могут возникать даже с самыми простыми (и на первый взгляд точными) денежными величинами, если хранить их в переменных типа `double`. Чтобы убедиться в этом, мы написали тестовую программу, которая определила переменную `d` следующим образом:

```
double d{123.02};
```

Затем мы вывели на экран значение `d`, отформатировав его до точности с `20` цифрами после точки. Оказалось, что в этом случае программа отображает число `123.02` как `123.0199999...` — это наглядный пример ошибки представления. Иногда денежные значения могут быть точно представлены в переменных `double`, а иногда неточно. Эта распространенная проблема есть во многих языках программирования. Далее мы покажем, как создать и использовать классы, позволяющие выполнять денежные расчеты идеально точно.

4.7. Инструкция цикла do...while

Напомним, что в инструкции `while` условие продолжения цикла проверяется перед выполнением тела цикла. Если это условие изначально ложно, тело цикла не будет выполнено ни разу. В инструкции `do...while` условие продолжения цикла проверяется после выполнения тела цикла, которое, таким образом, всегда выполняется хотя бы один раз. На илл. 4.5 показано, как числа от `1` до `10` выводятся в цикле `do...while`. В строке 7 объявляется и инициализируется управляющая переменная `counter`. В начале тела цикла `do...while` (строка 10) значение этой переменной выводится на экран, а в строке 11 оно увеличивается на `1`. В конце тела

цикла программа проверяет условие продолжения (строка 12). Если условие истинно, цикл продолжается с первой инструкции его тела (строка 10). Если условие ложно, цикл завершается и программа продолжает выполняться со следующей после цикла инструкции.

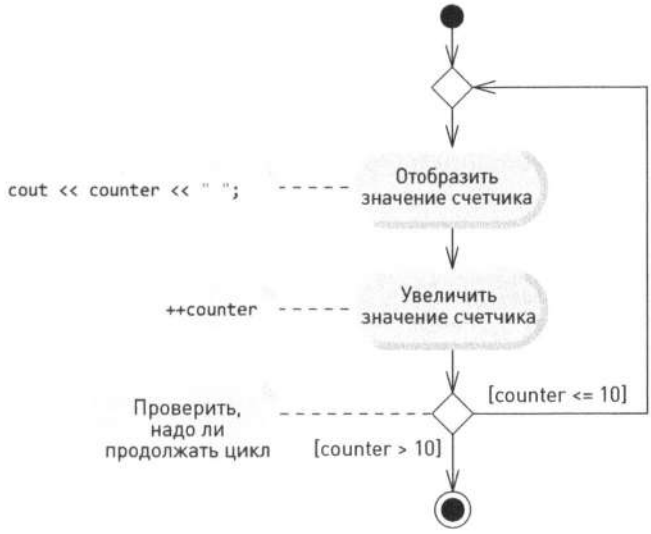
```
1 // fig04_05.cpp
2 // Цикл do...while
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7    int counter{1};
8
9    do {
10      cout << counter << " ";
11      ++counter;
12   } while (counter <= 10); // Конец цикла
13
14   cout << "\n";
15 }
```

```
1 2 3 4 5 6 7 8 9 10
```

Илл. 4.5 | Цикл do...while

UML-диаграмма цикла do...while

Диаграмма действий цикла `do...while` наглядно показывает, что условие выполнения цикла не будет проверяться, пока не будут выполнены хотя бы один раз инструкции в теле цикла:



4.8. Инструкция множественного выбора switch

Инструкция `switch` предназначена для выбора одного из нескольких альтернативных действий в зависимости от возможных значений переменной или выражения. Каждое из действий связано со значением **целочисленного константного выражения (integral constant expression)** — любой комбинации символьных и целочисленных констант, которая вычисляется как целочисленная константа.

Инструкция switch для подсчета оценок учеников в категориях A, B, C, D и F

На илл. 4.6 показан код, анализирующий введенные пользователем оценки учеников. Как и прежде, программа вычисляет среднюю оценку по классу. Кроме того, инструкция `switch` относит каждую оценку к одной из пяти категорий (`A`, `B`, `C`, `D` или `F`) и увеличивает соответствующий счетчик. Затем программа выводит данные о том, сколько учеников получили оценку в каждой из этих категорий.

```
// fig04_06.cpp
// Подсчет оценок учеников с помощью инструкции switch
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
   int total{0}; // Сумма оценок
   int gradeCounter{0}; // Счетчик введенных оценок
   int aCount{0}; // Счетчик оценок A
   int bCount{0}; // Счетчик оценок B
   int cCount{0}; // Счетчик оценок C
   int dCount{0}; // Счетчик оценок D
   int fCount{0}; // Счетчик оценок F

   cout << "Enter the integer grades in the range 0-100.\n"
      << "Type the end-of-file indicator to terminate input:\n"
      << " On UNIX/Linux/macOS type <Ctrl> d then press Enter\n"
      << " On Windows type <Ctrl> z then press Enter\n";

   int grade;

   // Цикл продолжается, пока пользователь не введет индикатор конца файла
   while (cin >> grade) {
      total += grade; // Прибавляем grade к total
      ++gradeCounter; // Увеличиваем счетчик введенных оценок

      // Увеличиваем счетчик оценок соответствующей категории
      switch (grade / 10) {
         case 9: // Оценка от 90
         case 10: // до 100 включительно
            ++aCount;
            break; // Выходим из switch
```

```
34
35          case 8: // Оценка от 80 до 89
36             ++bCount;
37             break; // Выходим из switch
38
39          case 7: // Оценка от 70 до 79
40             ++cCount;
41             break; // Выходим из switch
42
43          case 6: // Оценка от 60 до 69
44             ++dCount;
45             break; // Выходим из switch
46
47          default: // Оценка меньше 60
48             ++fCount;
49             break; // Не обязательно, так как мы все равно выходим из switch
50       } // Выходим из switch
51    } // Выходим из цикла
52
53    // Устанавливаем формат для чисел с плавающей точкой
54    cout << fixed << setprecision(2);
55
56    // Выводим заголовок отчета об оценках
57    cout << "\nGrade Report:\n";
58
59    // Если пользователь ввел хотя бы одну оценку...
60    if (gradeCounter != 0) {
61       // Вычисляем среднюю оценку по классу
62       double average{static_cast<double>(total) / gradeCounter};
63
64       // Выводим все результаты
65       cout << "Total of the " << gradeCounter << " grades entered is "
66          << total << "\nClass average is " << average
67          << "\nNumber of students who received each grade:"
68          << "\nA: " << aCount << "\nB: " << bCount << "\nC: " << cCount
69          << "\nD: " << dCount << "\nF: " << fCount << "\n";
70    }
71    else { // Не введена ни одна оценка, выводим сообщение об этом
72       cout << "No grades were entered" << "\n";
73    }
74 }
```

```
Enter the integer grades in the range 0-100.
Type the end-of-file indicator to terminate input:
   On UNIX/Linux/macOS type <Ctrl> d then press Enter
   On Windows type <Ctrl> z then press Enter
99
92
45
57
63
71
```

Окончание ⇨

```
76
85
90
100
^Z

Grade Report:
Total of the 10 grades entered is 778
Class average is 77.80

Number of students who received each grade:
A: 4
B: 1
C: 2
D: 1
F: 2
```

Илл. 4.6 | Подсчет оценок учеников с помощью инструкции switch

В строках 8 и 9 объявляются две локальные переменные: `total` для суммы введенных оценок и `gradeCounter` для их количества. В строках 10–14 объявляются и инициализируются значением `0` пять переменных для счетчиков оценок по категориям. В строках 24–51 программа получает от пользователя произвольное количество целочисленных оценок. Это происходит в цикле с контрольным значением, который обновляет `total` и `gradeCounter`, а также увеличивает счетчик оценок соответствующей категории. В строках 54–73 выводится отчет, содержащий количество введенных оценок, среднюю арифметическую оценку по классу и количество учеников, получивших оценки в каждой из категорий.

Ввод оценок пользователем

В строках 16–19 программа запрашивает у пользователя целочисленную оценку или **индикатор конца файла (end-of-file indicator)**, означающий, что пользователь завершил ввод данных. Индикатор конца файла — это стандартная комбинация клавиш, указывающая, что данные больше не будут вводиться. В главе 8 вы узнаете, как используется этот индикатор, когда программа получает данные из файла.

Какая комбинация клавиш служит индикатором конца файла, зависит от компьютера (от его операционной системы). Например, в UNIX, Linux и macOS нажмите комбинацию `Ctrl` + `d`. Эта запись означает, что клавиши `Ctrl` и `d` надо нажать одновременно. В Windows нажмите комбинацию `Ctrl` + `z`.

В некоторых системах требуется дополнительное нажатие клавиши `Enter`. Кроме того, Windows обычно отображает на экране символы `^Z`, когда вы вводите индикатор конца файла, как показано на илл. 4.6.

В строках 24–51 программа получает данные от пользователя с помощью цикла `while`. В строке 24 ввод данных происходит непосредственно в условии инструкции `while`:

```
while (cin >> grade) {
```

В этом случае условие считается истинным (`true`), если в поток `cin` успешно введено значение типа `int`. А если введен индикатор конца файла, условие считается ложным (`false`).

Если условие истинно, в строке 25 `grade` прибавляется к `total`, а в строке 26 увеличивается счетчик `gradeCounter`; потом эти данные понадобятся для вычисления среднего арифметического. В строках 29–50 инструкция `switch` увеличивает счетчик оценок в соответствующей категории.

Обработка оценок

Инструкция `switch` (строки 29–50) определяет, какой из пяти счетчиков надо увеличить. Мы предполагаем, что пользователь правильно ввел оценку: целое число от `0` до `100`. Мы распределили оценки по категориям таким образом: от `90` до `100` — `A`, от `80` до `89` — `B`, от `70` до `79` — `C`, от `60` до `69` — `D`, от `0` до `59` — `F`. Блок инструкции `switch` содержит несколько обозначенных **метками (labels)** секций `case` и необязательную секцию `default` (ее можно разместить в любом месте инструкции `switch`, но обычно ее размещают последней). Они определяют, какой из счетчиков надо увеличить в зависимости от введенной пользователем оценки.

Когда поток управления достигает инструкции `switch`, программа вычисляет **управляющее выражение (controlling expression)**, указанное в круглых скобках (`grade / 10`) после ключевого слова `switch`. Программа сравнивает значение этого выражения с каждой из меток `case`. Выражение должно иметь целочисленный тип со знаком или без знака: `bool`, `char`, `char8_t`, `char16_t`, `char32_t`, `wchar_t`, `int`, `long` или `long long`.

При вычислении управляющего выражения в строке 29 выполняется целочисленное деление, отсекающее дробную часть результата. При целочисленном делении на `10` любого числа от `0` до `100` результат обязательно будет в диапазоне от `0` до `10`. Мы используем несколько чисел из этого диапазона в метках `case`. Например, если пользователь ввел оценку `85`, управляющее выражение вычисляется как `8`. Инструкция `switch` сравнивает число `8` с каждой из меток `case`. Если найдено совпадение (`case 8:` в строке 35), выполняется инструкция после этой метки; в данном случае увеличивается счетчик `bCount` в строке 36, так как оценка в диапазоне от `80` до `89` входит в категорию `B`. Инструкция `break` (строка 37) завершает инструкцию `switch`. Здесь достигается конец цикла `while`, поэтому управление возвращается к проверке условия в строке 24, от которой зависит, будет ли повторен цикл.

В метках `case` инструкции `switch` проверяется совпадение с числами `10`, `9`, `8`, `7` и `6`. Обратите внимание на строки 30–31, в которых проверяются числа `9` и `10` (они относятся к категории `A`). Эта последовательность из двух меток `case` без каких-либо инструкций между ними приводит к одному и тому же результату в двух случаях: если управляющее выражение вычисляется как `9` или `10`, выполняются инструкции в строках 32–33. Инструкция `switch` не дает возможности проверять диапазоны чисел, поэтому каждое число, которое вам нужно проверить, должно быть указано в отдельной метке. Каждая секция `case` может выполнять несколько

перечисленных после нее инструкций, которые не требуется заключать в фигурные скобки (этим switch отличается от других управляющих инструкций), если вы не объявляете переменную внутри секции case.

Секция case без инструкции break и атрибут [[fallthrough]] (C++17)

При совпадении управляющего выражения с любой из меток case программа будет выполнять не только все инструкции из этой секции case, но и все инструкции из последующих секций case — до тех пор, пока их не прервет инструкция break или конец инструкции switch. Это называется **провалом (falling through)** в последующие секции case[1].

Отсутствие инструкции break, когда она необходима, — это логическая ошибка. Чтобы привлечь ваше внимание к возможной проблеме, многие компиляторы выдают предупреждение, если среди инструкций какой-либо секции case нет инструкции break. А для тех случаев, когда вы намеренно делаете провал в последующие секции case, в C++17 был введен **атрибут (attribute)** [[fallthrough]]. Если вы хотите сообщить компилятору, что отсутствие break — это правильное поведение программы, поместите вместо break следующую инструкцию:

```
[[fallthrough]];
```

Секция default

Если не найдено совпадения ни с одной из меток case, выполняется секция default (строки 47–49). В этом примере мы используем секцию default для обработки любых значений управляющего выражения, которые меньше 6, — все они относятся к категории F («экзамен не сдан»). Если не найдено совпадения ни с одной из меток case, а инструкция switch не содержит секции default, выполнение программы продолжается со следующей инструкции после switch. Опытный программист всегда проверяет в инструкции switch все возможные значения управляющего выражения, и секция default помогает не пропустить ни одно из них.

Вывод отчета об оценках

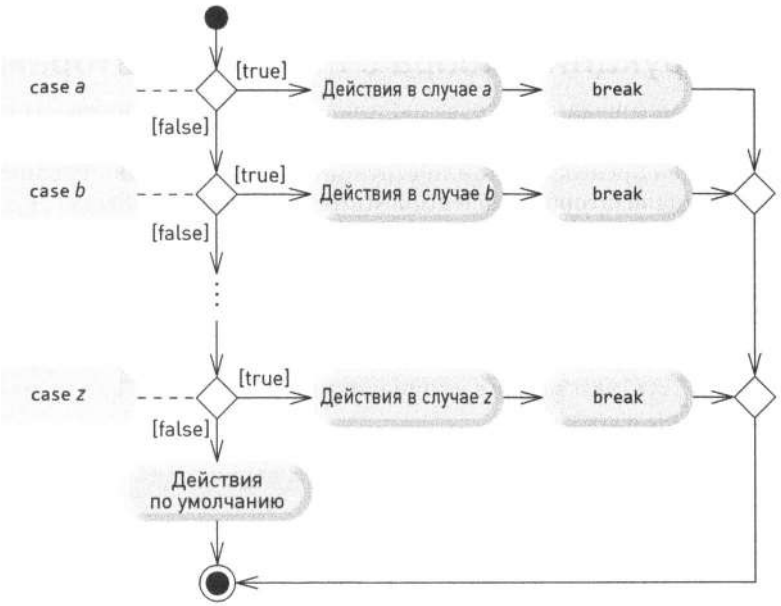
В строках 54–73 выводится отчет, основанный на введенных оценках. В строке 60 проверяется, ввел ли пользователь хотя бы одну оценку, — это предотвращает недопустимое деление на ноль. Целочисленное деление на ноль приводит к сбою программы, а результатом деления на ноль с плавающей точкой является специальное значение NaN — «Not a Number» («не число»). Если пользователь ввел хотя бы одну оценку, в строке 62 вычисляется среднее арифметическое. Затем в строках 65–69 выводится на экран: сумма оценок, средняя оценка по классу и ко-

[1] Это свойство инструкции switch хорошо подходит для простой программы, выводящей на экран текст известной песенки «Twelve Days of Christmas», в каждом куплете которой повторяется часть строк из предыдущего куплета. Попробуйте написать такую программу в качестве упражнения.

личество учеников, получивших оценки в каждой из буквенных категорий. Если оценки не вводились, в строке 72 отображается сообщение об этом. На илл. 4.6 показан пример отчета, основанного на десяти оценках.

UML-диаграмма инструкции switch

Ниже приведена обобщенная диаграмма действий для инструкции `switch`:



Почти всегда инструкция `switch` содержит в каждой из своих секций `case` инструкцию `break`, которая завершает `switch` после обработки этой секции. Это показано на диаграмме: каждая секция `case` завершается переходом к `break`, то есть управление немедленно передается на выход инструкции `switch`.

Инструкция `break` необязательна для самой последней секции (`case` или `default`), поскольку выполнение программы все равно продолжится со следующей инструкции после `switch`. Всегда включайте секцию `default` в каждую инструкцию `switch`, чтобы предусмотреть все варианты действий даже в исключительных условиях.

Метки case должны быть константами

Каждая секция `case` в инструкции `switch` должна содержать целочисленную константу (например, `8`, `9` или `10`) или целочисленное константное выражение (любое выражение, вычисляемое как целочисленная константа). Также можно использовать перечисления `enum` (см. раздел 5.9) и **символьные литералы (character literals)**, то есть определенные символы в одинарных кавычках (например, `'A'`, `'7'` или `'$'`), представляющие целочисленные значения символов. В приложе-

нии Б в конце книги приведены приведены целочисленные значения символов в кодировке ASCII, которая является подмножеством набора символов Unicode.

В главе 10 мы покажем вам, как более элегантно реализовать логику выбора нужного действия в зависимости от ситуации. Мы применим методику, называемую полиморфизмом, чтобы писать программы, которые нагляднее и проще для расширения функциональности, чем программы, в которых выбор осуществляется инструкциями `switch`.

4.9. Инструкции выбора с инициализаторами

Ранее мы рассматривали инструкцию цикла `for`. В ее заголовке мы объявили и инициализировали управляющую переменную, область видимости которой была ограничена пределами этой инструкции. В C++17 появились **инструкции выбора с инициализаторами (selection statements with initializers)** `if`, `if...else` и `switch`. Теперь вы можете инициализировать переменные перед условием в инструкциях `if` и `if...else`, а также перед управляющим выражением в инструкции `switch`. Как и в инструкции `for`, область видимости таких переменных ограничена пределами той инструкции, где они объявлены. На илл. 4.7 показаны примеры инструкций `if...else` с инициализацией. А в главе 5 (илл. 5.5) вы увидите, как инструкции `if...else` и `switch` с инициализацией используются в программе для популярной игры в кости.

```
1 // fig04_07.cpp
2 // Инструкции if с инициализаторами
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7    if (int value{7}; value == 7) {
8       cout << "value is " << value << "\n";
9    }
10   else {
11      cout << "value is not 7; it is " << value << "\n";
12   }
13
14   if (int value{13}; value == 9) {
15      cout << "value is " << value << "\n";
16   }
17   else {
18      cout << "value is not 9; it is " << value << "\n";
19   }
20 }
```

```
value is 7
value is not 9; it is 13
```

Илл. 4.7 | Инструкции if с инициализаторами

Синтаксис инструкций выбора с инициализаторами

В инструкциях `if` и `if...else` инициализатор переменной помещается в круглые скобки перед условием. В инструкции `switch` инициализатор также помещается в круглые скобки перед управляющим выражением. Инициализатор должен заканчиваться точкой с запятой (`;`), как показано в строках 7 и 14. Инициализатор может объявить несколько переменных одного типа в списке, разделенном запятыми.

Область видимости переменных, объявленных в инициализаторе

Любая переменная, объявленная в инициализаторе инструкции `if`, `if...else` или `switch`, видна только в пределах этой инструкции. Мы используем переменную `value` сначала в строках 7–12, чтобы определить, какую ветвь инструкции `if...else` надо выполнить, а затем — в инструкциях вывода обеих ветвей. Когда инструкция `if...else` завершается, эта переменная перестает существовать, поэтому мы можем снова использовать этот же идентификатор (имя `value`) в следующей инструкции `if...else`, чтобы объявить новую переменную, которая будет видна только в пределах этой инструкции.

Чтобы подтвердить недоступность `value` вне инструкций `if...else`, мы написали вторую версию этой программы (`fig04_07_with_error.cpp`), которая пытается получить доступ к переменной `value` после завершения второй инструкции `if...else` (после — значит вне области видимости). Это приводит к следующим ошибкам компиляции:

- Visual Studio: `'value': undeclared identifier`
- Xcode: `error: use of undeclared identifier 'value'`
- GNU g++: `error: 'value' was not declared in this scope`

4.10. Инструкции break и continue

В дополнение к инструкциям выбора и циклов в C++ есть инструкции `break` и `continue`, предназначенные для изменения потока управления. В предыдущем разделе было показано, как `break` прерывает выполнение инструкции `switch`. В этом разделе мы обсудим, как использовать `break` в инструкциях циклов.

Инструкция break

Выполнение инструкции `while`, `for`, `do...while` или `switch` немедленно прерывается (завершается), как только выполняется вложенная в нее инструкция `break`. Программа продолжит выполняться со следующей инструкции. Обычно `break` применяют для раннего выхода из цикла или из инструкции `switch` (как на илл. 4.6). На илл. 4.8 показан пример раннего завершения цикла `for` с помощью `break`.

```
1 // fig04_08.cpp
2 // Прерывание цикла for инструкцией break
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7    int count; // Управляющая переменная, также используемая после цикла
8
9    for (count = 1; count <= 10; ++count) { // Повторяем цикл 10 раз
10      if (count == 5) {
11         break; // Но прерываем цикл, если значение count равно 5
12      }
13
14      cout << count << " ";
15   }
16
17   cout << "\nBroke out of loop at count = " << count << "\n";
18 }
```

```
1 2 3 4
Broke out of loop at count = 5
```

Илл. 4.8 | Прерывание цикла for инструкцией break

Когда инструкция `if` (строки 10–12), вложенная в цикл `for` (строки 9–15), обнаруживает, что значение `count` равно `5`, выполняется инструкция `break` в строке 11. Это прерывает цикл `for`. Программа переходит к строке 17 (следующая инструкция после цикла `for`), в которой выводится сообщение со значением переменной `count` после завершения цикла. Тело цикла полностью выполняется только четыре раза вместо десяти. Обратите внимание, что мы могли бы инициализировать `count` в строке 7 и оставить пустой секцию инициализации заголовка цикла `for`:

```
for (; count <= 10; ++count) { // Повторяем цикл 10 раз
```

Инструкция continue

Как только выполняется инструкция `continue`, вложенная в тело цикла `while`, `for` или `do...while`, все последующие инструкции в теле цикла пропускаются и начинается следующая итерация цикла. В инструкциях `while` и `do...while` условие продолжения цикла проверяется сразу после выполнения инструкции `continue`. В инструкции `for` сначала выполняется инкремент управляющей переменной, а затем проверяется условие продолжения цикла.

```
1 // fig04_09.cpp
2 // Прерывание текущей итерации цикла for инструкцией continue
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7    for (int count{1}; count <= 10; ++count) { // Повторяем цикл 10 раз
8       if (count == 5) {
```

```
 9          continue; // Но прерываем цикл, если значение count равно 5
10       }
11
12       cout << count << " ";
13    }
14
15    cout << "\nUsed continue to skip printing 5" << "\n";
16 }
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

Илл. 4.9 | Прерывание текущей итерации цикла for инструкцией continue

Инструкция `continue` (строка 9) используется для того, чтобы пропустить строку 12, когда вложенная инструкция `if` обнаружит, что значение `count` равно `5`. Сразу после инструкции `continue` выполнение программы продолжится с инкремента управляющей переменной в инструкции `for` (строка 7).

Некоторые программисты считают, что `break` и `continue` нарушают принципы структурного программирования. Поскольку аналогичные переходы можно реализовать методами структурного программирования, эти программисты избегают инструкций `break` и `continue`.

Требования к высокому качеству программного продукта часто вступают в противоречие с требованиями к его быстродействию. Иногда одна из этих целей достигается в ущерб другой. В любой ситуации (особенно при создании высоконагруженных приложений) старайтесь прежде всего делать код простым и правильным. И только после этого приступайте к оптимизации кода, чтобы сделать его быстрее и компактнее, — но только если это действительно необходимо.

4.11. Логические операторы

Условия в инструкциях `if`, `if...else`, `while`, `do...while` и `for` определяют, каким будет поток управления программы. До сих пор мы использовали только простые условия, например: `count <= 10`, `number != sentinelValue` или `total > 1000`. Простые условия выражаются с помощью операторов сравнения `>`, `<`, `>=`, `<=` и операторов равенства `==` и `!=`. Каждый из них проверяет только одно условие. Иногда в управляющих инструкциях нужны более сложные условия, от которых зависит поток управления. **Логические операторы (logical operators)** языка C++ позволяют составлять комбинации из простых условий. Логическими операторами являются: `&&` (логическое И), `||` (логическое ИЛИ) и `!` (логическое НЕ).

4.11.1. Оператор && (логическое И)

Предположим, что в какой-то точке программы мы хотим проверить истинность двух условий одновременно, чтобы выбрать следующее действие.

Для этого нам понадобится оператор && — **логическое И (logical AND)**:

```
if (gender == FEMALE && age >= 65) {
   ++seniorFemales;
}
```

Предположим, что `FEMALE` — это константа (константная переменная). Инструкция `if` содержит два простых условия. Условие `gender == FEMALE` определяет, является ли человек женщиной. Условие `age >= 65` определяет, является ли человек пожилым. Оператор `&&` объединяет два простых условия в одно составное:

```
gender == FEMALE && age >= 65
```

Оно будет истинно, если истинны оба простых условия одновременно. В этом случае инструкция в теле `if` увеличивает переменную `seniorFemales` на `1`. Если же хотя бы одно из простых условий является ложным, программа пропускает инструкцию в теле `if`. Некоторые программисты рекомендуют добавлять для наглядности необязательные круглые скобки:

```
(gender == FEMALE) && (age >= 65)
```

Следующая **таблица истинности (truth table)** показывает все четыре комбинации логических значений для двух выражений, объединенных оператором `&&`. Любое выражение, содержащее операторы равенства, операторы сравнения или логические операторы, вычисляется компилятором C++ как `false` (ноль) или `true` (любое значение, кроме нуля).

Выражение 1	Выражение 2	Выражение 1 && Выражение 2
`false`	`false`	`false`
`false`	`true`	`false`
`true`	`false`	`false`
`true`	`true`	`true`

4.11.2. Оператор || (логическое ИЛИ)

Теперь предположим, что мы хотим проверить истинность как минимум одного из двух условий, чтобы выбрать следующее действие. Для этого нам понадобится оператор || — **логическое ИЛИ (logical OR)**:

```
if ((semesterAverage >= 90) || (finalExam >= 90)) {
   cout << "Student grade is A\n";
}
```

Здесь два простых условия тоже объединены в одно составное. Условие `semesterAverage >= 90` определяет, заслуживает ли студент пятерки по курсу за

хорошую успеваемость в течение семестра. Условие `finalExam >= 90` определяет, заслуживает ли студент пятерки по курсу за выдающийся результат на итоговом экзамене. Инструкция `if` проверяет составное условие:

```
(semesterAverage >= 90) || (finalExam >= 90)
```

и присуждает студенту оценку A (пятерку), если выполняется одно или оба простых условия. Сообщение `Student grade is A` не будет выведено, если оба простых условия окажутся ложными. Ниже приведена таблица истинности для двух выражений, объединенных оператором `||`.

Выражение 1	Выражение 2	Выражение 1 || Выражение 2
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`true`

Оператор `&&` имеет более высокий приоритет, чем оператор `||`[1]. Оба оператора группируются слева направо.

4.11.3. Вычисление по короткой схеме

Простые условия, объединенные операторами `&&` или `||` в выражение, вычисляются слева направо только до тех пор, пока программа еще не выяснила, является ли это выражение истинным или ложным. Например, вычисление этого выражения сразу прекращается, если первое условие (`gender == FEMALE`) оказалось ложным (в этом случае все выражение тоже будет ложным):

```
(gender == FEMALE) && (age >= 65)
```

Если же первое условие оказалось истинным, то продолжается вычисление второго условия (`age >= 65`). Если и оно окажется истинным, то все выражение тоже будет истинным. Этот способ вычисления выражений с операторами `&&` и `||` называется **вычислением по короткой схеме (short-circuit evaluation)**.

В некоторых выражениях с оператором `&&` одно из условий (мы будем называть его зависимым условием) может требовать, чтобы другое условие было истинным; иначе вычисление зависимого условия может оказаться невозможным. В таких случаях надо поместить зависимое условие после оператора `&&`, чтобы предотвратить ошибку. Например, рассмотрим выражение `(i != 0) && (10 / i == 2)`. Мы поместили зависимое условие `(10 / i == 2)` после оператора `&&`, чтобы исключить

[1] Всегда используйте круглые скобки, если без них возможна двусмысленность в отношении порядка вычислений.

возможность деления на ноль (второе условие не будет вычисляться, если первое окажется ложным).

4.11.4. Оператор ! (логическое НЕ)

Теперь рассмотрим оператор ! — **логическое НЕ (logical NOT)**, или **логическое отрицание (logical negation)**. Он меняет значение условия на противоположное. В отличие от логических операторов **&&** и **||**, которые являются бинарными (объединяющими два условия), логический оператор ! является унарным (он имеет только одно условие в качестве операнда). Чтобы выполнить код только тогда, когда условие ложно (`false`), поместите перед этим условием оператор логического отрицания:

```
if (!(grade == sentinelValue)) {
   cout << "The next grade is " << grade << "\n";
}
```

В этом примере инструкция в теле `if` выполняется, если `grade` не равно `sentinelValue`. Круглые скобки вокруг условия `grade == sentinelValue` необходимы, поскольку оператор логического отрицания имеет более высокий приоритет, чем оператор равенства.

В большинстве случаев вы можете обойтись без логического отрицания, выразив то же самое условие с помощью подходящего оператора равенства или сравнения. Например, предыдущую инструкцию можно записать другим способом:

```
if (grade != sentinelValue) {
   cout << "The next grade is " << grade << "\n";
}
```

Такая гибкость помогает выражать любое условие наиболее понятно. Ниже приведена таблица истинности для оператора логического отрицания.

Выражение	!Выражение
`false`	`true`
`true`	`false`

4.11.5. Программное создание таблиц истинности для логических операторов

На илл. 4.10 показана программа, использующая логические операторы для создания таблиц истинности. Она выводит каждое вычисленное выражение и его результат типа `bool`. По умолчанию оба значения типа `bool` (`true` и `false`) выводятся через `cout` как `1` и `0` соответственно, но функция `format` позволяет выводить их в виде слов `true` и `false`. В строках 10–14, 17–21 и 24–26 формируются таблицы истинности для операторов **&&**, **||** и **!** соответственно.

```
// fig04_10.cpp
// Логические операторы
#include <iostream>
#include <fmt/format.h> // Или в C++20: #include <format>
using namespace std;
using namespace fmt; // Не требуется в C++20

int main() {
   // Составляем таблицу истинности для оператора && (логическое И)
   cout << "Logical AND (&&)\n"
      << format("false && false: {}\n", false && false)
      << format("false && true: {}\n", false && true)
      << format("true && false: {}\n", true && false)
      << format("true && true: {}\n\n", true && true);

   // Составляем таблицу истинности для оператора || (логическое ИЛИ)
   cout << "Logical OR (||)\n"
      << format("false || false: {}\n", false || false)
      << format("false || true: {}\n", false || true)
      << format("true || false: {}\n", true || false)
      << format("true || true: {}\n\n", true || true);

   // Составляем таблицу истинности для оператора ! (логическое НЕ)
   cout << "Logical negation (!)\n"
      << format("!false: {}\n", !false)
      << format("!true: {}\n", !true);
}
```

```
Logical AND (&&)
false && false: false
false && true: false
true && false: false
true && true: true

Logical OR (||)
false || false: false
false || true: true
true || false: true
true || true: true

Logical negation (!)
!false: true
!true: false
```

Илл. 4.10 | Логические операторы

Приоритеты и правила группировки операторов

В следующей таблице показаны приоритеты и правила группировки всех представленных на данный момент операторов C++. Они перечислены сверху вниз в порядке убывания приоритетов.

Операторы[1]						Группировка (ассоциативность)
++	--	static_cast<*тип*>			()	Слева направо
++	--	+	-	!		Справа налево
*	/	%				Слева направо
+	-					Слева направо
<<	>>					Слева направо
<	<=	>	>=			Слева направо
==	!=					Слева направо
&&						Слева направо
||						Слева направо
?:						Справа налево
=	+=	-=	*=	/=	%=	Справа налево
,						Слева направо

4.12. Типичные ошибки при работе с операторами равенства (==) и присваивания (=)

Есть логические ошибки, которые даже у опытных программистов C++ случаются так часто, что заслуживают отдельного раздела. Внешне похожие операторы равенства (==) и присваивания (=) легко перепутать, и эти опечатки усугубляются тем, что при случайной замене одного из операторов другим обычно не возникает ошибка компиляции. Инструкции с такими опечатками могут компилироваться и выполняться, но из-за них программа работает с логическими ошибками. Современные компиляторы могут выдавать предупреждения, когда программист ставит оператор присваивания (=) там, где ожидается ==. В конце раздела вы узнаете, как включить эту функцию.

Названная проблема обусловлена двумя особенностями языка C++. Во-первых, любое вычисляемое выражение может использоваться в секции принятия решения любой управляющей инструкции. Если значение выражения равно `0`, оно обрабатывается как `false`. Если значение не равно `0`, оно обрабатывается как `true`. Во-вторых, при присваивании создается значение, а именно значение переменной в левой части оператора присваивания. Предположим, вы хотели написать:

```
if (payCode == 4) { // Правильно
   cout << "You get a bonus!" << "\n";
}
```

[1] Обратите внимание, что операторы могут выглядеть совершенно одинаково, но при этом отличаться по своему действию, по приоритету и по ассоциативности. Приложение А в конце книги поможет вам уверенно различать такие операторы. — *Примеч. ред.*

но случайно перепутали операторы и написали:

```
if (payCode = 4) { // Неправильно
   cout << "You get a bonus!" << "\n";
}
```

Первая инструкция `if` присуждает бонус человеку, чей `payCode` равен 4. Вторая инструкция `if` (неправильная) вычисляет выражение присваивания переменной в условии `if`, получая в результате число 4. Любое ненулевое значение эквивалентно `true`, поэтому это условие всегда истинно и человек получает бонус независимо от `payCode`. Что еще хуже, неправильная инструкция изменила значение переменной `payCode`, которое не должно было измениться.

«Левые» (lvalue) и «правые» (rvalue) значения

Вы можете избежать этой проблемы, применяя простое правило. Прежде всего, запомните, что переменную обычно ставят слева от оператора присваивания. Переменную в выражении присваивания называют термином **lvalue** (**left value** — **значение слева**). А литерал (константу) всегда ставят справа от оператора присваивания. Литерал в выражении присваивания называют термином **rvalue** (**right value** — **значение справа**). При необходимости переменную (то есть *lvalue*) можно поставить и справа от оператора присваивания, но литерал (то есть *rvalue*) — только справа!

Многие программисты таким же образом записывают и условия равенства: слева от оператора равенства ставят переменную, а справа — литерал (например, `x == 7`). Но лучше поступать наоборот: ставьте литерал слева от оператора равенства, а переменную — справа (например, `7 == x`). Эта форма записи совершенно верна синтаксически, иногда ее называют «условием Йоды»[1]. Если вы будете следовать этому правилу и однажды случайно замените оператор равенства (`==`) оператором присваивания (`=`), компилятор сообщит вам об ошибке. Ошибка компиляции возникнет из-за того, что получится недопустимое выражение присваивания с литералом слева.

Ошибка: равенство (==) вместо присваивания (=)

Часто возникает другая неприятная ситуация. Предположим, вы хотели присвоить значение переменной:

```
x = 1;
```

но случайно перепутали операторы и написали:

```
x == 1;
```

Это тоже не считается синтаксической ошибкой. Компилятор просто выполнит вычисление этого выражения. Если `x` равно `1`, то условие истинно; результатом выражения будет ненулевое значение (`true`). Если `x` не равно `1`, то условие ложно; результатом выражения будет ноль (`false`). Каким бы ни было значение выраже-

[1] «Yoda conditions». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Yoda_conditions.

ния, оно будет потеряно. При этом переменная x не изменится, что приведет к логической ошибке во время выполнения программы. Использование оператора == для присваивания и оператора = для равенства — это логические ошибки. Для проверки вашего кода откройте его в текстовом редакторе, найдите все символы = и убедитесь, что все операторы присваивания и равенства написаны правильно.

Включение предупреждений

Xcode автоматически выдаст предупреждение, если вы поставите оператор = там, где ожидается ==. В некоторых компиляторах такие предупреждения по умолчанию отключены. Если вы работаете с компилятором GNU g++, добавьте флаг `-Wall` (включение всех предупреждений) в вашу команду компиляции. Вы также можете включить лишь некоторые виды предупреждений, руководствуясь документацией g++. Если вы работаете в Visual C++:

1. В вашем решении (`Solution`) щелкните правой кнопкой мыши на имени проекта и выберите `Properties`.
2. Разверните раздел `Code Analysis` и выберите пункт `General`.
3. Выберите `Yes` для `Enable Code Analysis on Build` и нажмите `OK`.

4.13. Готовые объекты: создание и чтение ZIP-файлов[1]

Сжатие данных (data compression) уменьшает объем данных: как правило, это нужно для экономии дискового пространства или для ускорения передачи данных через интернет путем уменьшения количества передаваемых байтов. **Алгоритмы сжатия данных без потерь (lossless data-compression algorithms)** работают так, что сжатые данные всегда можно восстановить в исходном виде, без малейших искажений. **Алгоритмы сжатия данных с потерями (lossy data-compression algorithms)** работают так, что какая-то часть данных безвозвратно теряется (удаляется). Такими алгоритмами обычно сжимают изображения, фильмы и музыку. Например, потоковое видео, которое вы смотрите в браузере, заранее сжато с потерями, поэтому объем передаваемых браузеру данных сведен к минимуму. Но благодаря хорошему алгоритму, после такого сжатия большинство зрителей не замечает ухудшения изображения. Качество видео остается достаточно хорошим.

ZIP-файлы

Скорее всего, вы знакомы с ZIP-файлами. Если нет, то сейчас познакомитесь. Формат ZIP-файла[2] — это формат сжатия без потерь[3], который используется уже более 30 лет. В алгоритмах сжатия без потерь сочетаются многие методы, в том числе:

---

[1] Пример исходного кода в этом разделе несовместим с компилятором GNU g++.

[2] «ZIP». Был доступен: 15 сентября 2023 г. https://ru.wikipedia.org/wiki/ZIP.

[3] «Сжатие без потерь». Был доступен: 15 сентября 2023 г. https://ru.wikipedia.org/wiki/Сжатие_без_потерь.

- замена многих повторяющихся фрагментов (например, слов в тексте или пикселей в изображении) ссылками на единственный экземпляр каждого такого фрагмента;
- замена в изображении группы пикселей, имеющих один и тот же цвет, на один пиксель этого цвета с указанием количества одинаковых пикселей в группе (метод кодирования повторов).

Технология ZIP позволяет сжимать (архивировать) многочисленные файлы и папки в один файл, который называется **архивным файлом (archive file)** или просто архивом. Дистрибутивы программ часто распространяются в интернете в виде ZIP-файлов для ускорения загрузки. Современные операционные системы обычно имеют встроенную поддержку ZIP-формата, то есть могут создавать ZIP-архивы и извлекать их содержимое.

Библиотека miniz-cpp с открытым исходным кодом

Многие библиотеки с открытым исходным кодом могут добавить в ваши программы поддержку ZIP-формата, а также других популярных архивных форматов, например TAR, RAR и 7-Zip[1]. Программа, показанная на илл. 4.11, продолжает нашу презентацию готовых объектов: теперь мы будем работать с объектами библиотеки `miniz-cpp` с открытым исходным кодом[2,3], которая позволяет создавать и распаковывать ZIP-файлы. Это **заголовочная библиотека (header-only library)**, то есть она полностью определена в заголовочном файле `zip_file.hpp`. Для работы с ней достаточно скопировать этот файл в папку с исходным кодом программы и включить директивой `#include` (строка 5). Мы уже поместили библиотеку в папку `libraries/miniz-cpp`, которая находится в папке `examples`. Заголовки подробно рассматриваются в главе 9.

Для удобства обсуждения код на илл. 4.11 разделен на фрагменты с примерами ввода и вывода. Для краткости мы поставили подпись только под первым фрагментом и в других аналогичных примерах будем делать так же.

```
1 // fig04_11.cpp
2 // Создание и чтение ZIP-файла с помощью библиотеки miniz-cpp
3 #include <iostream>
4 #include <string>
5 #include "zip_file.hpp"
6 using namespace std;
7
```

Илл. 4.11 | Создание и чтение ZIP-файла с помощью библиотеки miniz-cpp

[1] «List of archive formats». 19 марта 2020 г. https://en.wikipedia.org/wiki/List_of_archive_formats.

[2] https://github.com/tfussell/miniz-cpp.

[3] Библиотека miniz-cpp дает примерно такие же возможности, как модуль zipfile стандартной библиотеки Python (https://docs.python.org/3/library/zipfile.html), поэтому репозиторий miniz-cpp на портале GitHub отсылает вас к этой странице документации.

Функция getline для ввода текста

Вызываемая в строке 11 функция `getline` считывает все введенные с клавиатуры символы, пока вы не нажмете Enter:

```
 8 int main() {
 9    cout << "Enter a ZIP file name: ";
10    string zipFileName;
11    getline(cin, zipFileName); // Вводим строку текста
12
```

```
Enter a ZIP file name: c:\users\useraccount\Documents\test.zip
```

Здесь мы используем функцию `getline` для получения от пользователя строки, указывающей путь к ZIP-файлу и имя этого файла. Полученная строка сохраняется в переменной `zipFileName` типа `string`. Как и класс `string`, функция `getline` требует заголовка `<string>` и находится в пространстве имен `std`.

Создание образца контента для последующего сжатия в ZIP-архив

Следующая инструкция создает длинную строку с именем `content`, содержащую фрагмент текста из начала этой главы:

```
13    // Строковые литералы, разделенные только пробелами,
14    // компилятор объединяет в одну строку
15    string content{
16       "This chapter introduces all but one of the remaining control "
17       "statements--the for, do...while, switch, break and continue "
18       "statements. We explore the essentials of counter-controlled "
19       "iteration. We use compound-interest calculations to begin "
20       "investigating the issues of processing monetary amounts. First, "
21       "we discuss the representational errors associated with "
22       "floating-point types. We use a switch statement to count the "
23       "number of A, B, C, D and F grade equivalents in a set of "
24       "numeric grades. We show C++17's enhancements that allow you to "
25       "initialize one or more variables of the same type in the "
26       "headers of if and switch statements."};
27
```

С помощью библиотеки `miniz-cpp` мы запишем эту строку в текстовый файл, помещенный в ZIP-архив. Каждый строковый литерал в этой инструкции отделен от следующего только пробелом. Компилятор C++ автоматически объединяет такие строковые литералы в одну строку, и мы инициализируем этой строкой переменную `content`. Следующая инструкция выводит на экран длину переменной `content` (632 байта).

```
28    cout << "\ncontent.length(): " << content.length();
29
```

```
content.length(): 632
```

Создание объекта zip_file

Класс `zip_file` библиотеки `miniz-cpp`, расположенный в пространстве имен `miniz_cpp`, используется для создания ZIP-файлов. Следующая инструкция создает объект класса `zip_file` с именем `output`:

```
30    miniz_cpp::zip_file output; // Создаем объект zip_file
31
```

Этот объект будет выполнять все операции для создания ZIP-файла.

Создание файла в объекте zip_file и сохранение этого объекта на диск

В строке 33 вызывается функция `writestr` объекта `output`, которая помещает в ZIP-архив файл `intro.txt`, содержащий текст из переменной `content`. В строке 34 вызывается функция `save` объекта `output`, которая сохраняет содержимое объекта `output` в файл, указанный в переменной `zipFileName`.

```
32    // Записываем строку content в текстовый файл в объекте output
33    output.writestr("intro.txt", content); // Создаем ZIP-архив
34    output.save(zipFileName); // Сохраняем output в ZIP-файле zipFileName
35
```

ZIP-файл похож на набор случайных символов

ZIP — это бинарный формат, поэтому вы не сможете прочесть текст в сжатом файле, если откроете его в текстовом редакторе. Ниже показано, как выглядит созданный нами файл в «Блокноте» Windows:

```
PK    ♦ ♦t—P♦VE´o♦  y          intro.txt]'KnU0♣†÷>♦±
tõu›"'(ÐS-Ó#¢²ä'ÒL'ÇÈ‰CÊƒ ÈÒ&éï¥øwd…Ñ¨$À¹JY[ ♦L  -V¡d,²A♦B;ræ|P|1
Z±ÒN¹ê4ú£Vd„µÌó|‹æh%q
q„Eÿ æµŸrnôév†?löÿHE¨ƒHÕÞ3&ut(-› ,é♦M´♦♦ÌX¹ä~Ú"li?loФwI+♦L¡¥%¥P♦,tá<p%Ú/öÞ|8ŒU0uÐ!Åæ«♦v³]Qî
€»ÓM♦♦‹Öq ♦¬¬¡@›"9♣fb;♦♦H♦KO♦♦6ƒ+X qØRéÄé(&♦êý ýÐŽ˜à#0«Úm;dÈm_¬Søc„Ÿ#<ð«gù ♦ÁÕòú×ØŠÉ³´
ý(dÅm~HÂáÜ:y♦Ë
žŸž¾}¤S 1‡³♦#aõÖm~/ÍÐXV♦1%ð+♦X,ÖÐ♦L.qIgf,âNÝ'Ã]p$âIŸóÖ•~5"óß♦PK♦  ♦  ♦ ♦t—P♦VÉ´o♦  y
intro.txtPK♦  ♦ ♦ 7  -♦
```

Чтение содержимого ZIP-файла

Вы можете открыть созданный ZIP-файл на своем компьютере и просмотреть (извлечь) его содержимое средствами операционной системы. Так вы убедитесь, что архив был создан без ошибок. Но и библиотека `miniz-cpp` тоже позволяет открывать ZIP-файлы. Следующая инструкция создает объект класса `zip_file` с именем `input` и инициализирует его содержимым переменной `zipFileName`, то есть именем ZIP-файла.

```
36    miniz_cpp::zip_file input(zipFileName); // Открываем файл zipFileName
37
```

В результате содержимое соответствующего ZIP-файла считывается в объект `zip_file`. Теперь мы можем использовать функции этого объекта для взаимодействия с файлом архива.

Вывод на экран имени и содержимого ZIP-файла

Следующие инструкции вызывают функции `get_filename` и `printdir` объекта `input` для вывода имени ZIP-файла и его содержимого соответственно.

```
38    // Выводим на экран имя и содержимое ZIP-файла
39    cout << "\n\nZIP file's name: " << input.get_filename()
40       << "\n\nZIP file's directory listing:\n";
41    input.printdir();
42
```

```
ZIP file's name: c:\users\useraccount\Documents\test.zip

ZIP file's directory listing:
  Length      Date    Time    Name
---------  ---------- -----   ----
      632  11/28/2021 16:48   intro.txt
---------                     -------
      632                     1 file
```

Вы видите, что ZIP-архив содержит файл `intro.txt` длиной 632 байта, что соответствует длине строки `content`, записанной в этот файл.

Вывод на экран информации о конкретном файле из ZIP-архива

В строке 44 объявляется и инициализируется объект класса `zip_info` с именем `info`:

```
43    // Выводим информацию о сжатом файле intro.txt
44    miniz_cpp::zip_info info{input.getinfo("intro.txt")};
45
```

Вызов функции `getinfo` объекта `input` возвращает объект `zip_info` (из пространства имен `miniz_cpp`), соответствующий файлу `intro.txt` из архива. Иногда объекты предоставляют данные таким образом, что вы можете получить к ним прямой доступ, используя имя объекта и оператор «точка» (`.`). Например, объект `info` предоставляет данные о находящемся в архиве файле `intro.txt`: его имя (`info.filename`), исходный размер (`info.file_size`) и размер в сжатом виде (`info.compress_size`):

```
46    cout << "\nFile name: " << info.filename
47       << "\nOriginal size: " << info.file_size
48       << "\nCompressed size: " << info.compress_size;
49
```

```
File name: intro.txt
Original size: 632
Compressed size: 360
```

Обратите внимание, что размер `intro.txt` в сжатом виде — всего `360` байт, то есть на 43% меньше, чем исходный размер файла. Степень сжатия алгоритмом ZIP варьируется в больших пределах в зависимости от вида сжимаемого контента.

Извлечение файла «intro.txt» из архива

Чтобы извлечь из ZIP-архива исходное содержимое сжатого файла, мы используем функцию `read` объекта `input`, передавая ей в качестве аргумента объект класса `zip_info` с именем `info`. Эта функция возвращает строку с исходным содержимым файла, представленного объектом `info`:

```
50    // Исходное содержимое файла
51    string extractedContent{input.read(info)};
52
```

Затем мы выводим на экран полученную строку `extractedContent`, чтобы показать ее совпадение с исходной строкой `content`, которую мы поместили в архив. Как видите, это действительно было сжатие без потерь:

```
53    cout << "\n\nOriginal contents of intro.txt:\n"
54       << extractedContent << "\n";
55 }
```

```
Original contents of intro.txt:
This chapter introduces all but one of the remaining control statements--the for,
do...while, switch, break and continue statements. We explore the essentials of
counter-controlled iteration. We use compound-interest calculations to begin
investigating the issues of processing monetary amounts. First, we discuss the
representational errors associated with floating-point types. We use a switch
statement to count the number of A, B, C, D and F grade equivalents in a set of
numeric grades. We show C++17's enhancements that allow you to initialize one or
more variables of the same type in the headers of if and switch statements.
```

4.14. Форматирование текста с указанием ширины полей и точности

В разделе 3.13 вы познакомились с появившейся в C++20 функцией `format` (заголовок `<format>`), которая расширила возможности форматирования текста. Сейчас вы увидите, что в форматируемых строках можно указывать, как надо отформатировать каждое конкретное значение. Мы по-новому реализуем такое же форматирование, как в задаче о сложных процентах (см. илл. 4.4). Новая версия программы выводит те же данные, что и прежняя, поэтому мы сосредоточимся исключительно на форматировании в строках 13, 14, 17 и 22.

```
1 // fig04_12.cpp
2 // Расчет сложных процентов и форматирование текста
3 #include <iostream>
```

Окончание ⇨

```
#include <cmath> // Для функции возведения в степень
#include <fmt/format.h> // Или в C++20: #include <format>
using namespace std;
using namespace fmt; // Не требуется в C++20

int main() {
   double principal{1000.00}; // Первоначальный вклад
   double rate{0.05}; // Процентная ставка

   cout << format("Initial principal: {:>7.2f}\n", principal)
      << format(" Interest rate: {:>7.2f}\n", rate);

   // Выводим заголовки таблицы
   cout << format("\n{}{:>20}\n", "Year", "Amount on deposit");

   // Рассчитываем сумму на счете в конце каждого года из десяти
   for (int year{1}; year <= 10; ++year) {
      double amount = principal * pow(1.0 + rate, year);
      cout << format("{:>4d}{:>20.2f}\n", year, amount);
   }
}
```

```
Initial principal: 1000.00
Interest rate:        0.05

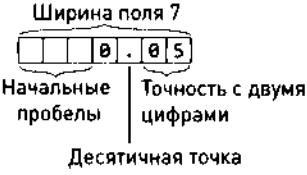
Year Amount on deposit
   1           1050.00
   2           1102.50
   3           1157.63
   4           1215.51
   5           1276.28
   6           1340.10
   7           1407.10
   8           1477.46
   9           1551.33
  10           1628.89
```

Илл. 4.12 | Расчет сложных процентов и форматирование текста в C++20

Форматирование значений суммы и процентной ставки

В вызовах функции `format` в строках 13 и 14 используется заполнитель `{:>7.2f}`, форматирующий значения переменных `principal` и `rate`. Двоеточие (`:`) в заполнителе предшествует **спецификатору формата (format specifier)**, который указывает, как надо отформатировать соответствующее значение. Спецификатор формата `>7.2f` означает, что число с плавающей точкой (`f`) должно быть **выровнено по правому краю (right-aligned)** (`>`) в поле шириной 7 символов и иметь две цифры (`.2`) после точки. В отличие от известных вам манипуляторов потока `setprecision` и `fixed`, указанные в этих заполнителях параметры формата не являются липкими и применяются только к тому значению, которым будет заменен заполнитель.

Отображаемое значение переменной `principal` (`1000.00`) занимает семь символов, то есть всю ширину поля, поэтому для его выравнивания по ширине поля не нужны пробелы. Отображаемое значение переменной `rate` (`0.05`) занимает четыре символа, поэтому программа выводит его в поле из семи символов с выравниванием по правому краю, а левую часть этого поля заполняет тремя пробелами:



Числа по умолчанию выравниваются по правому краю поля, поэтому символ `>` в данном случае необязателен. Вы можете выравнивать числовые значения по левому краю поля с помощью символа `<`.

Форматирование заголовков таблицы

Для вывода заголовков в строке 17 используется следующая строка формата:

```
"\n{}{:>20}\n"
```

Она означает, что заголовок `"Year"` помещается в позицию первого заполнителя, который не содержит спецификатора формата. Второй заполнитель указывает, что заголовок `"Amount on deposit"` (в нем 17 символов) надо выровнять по правому краю (`>`) в поле шириной `20` символов; для этого выравнивания функция `format` вставляет три пробела перед заголовком. Строки по умолчанию выравниваются по левому краю поля, поэтому символ `>` в данном случае обязателен.

Форматирование данных таблицы в цикле for

В строке 22 используется следующая строка формата:

```
"{:>4d}{:>20.2f}\n"
```

В ней есть два заполнителя, форматирующие выполняемый в цикле вывод данных. Заполнитель `{:>4d}` указывает, что значение переменной `year` надо отформатировать как целое число (`d` означает «decimal integer»), выровненное по правому краю (`>`) в поле шириной `4` символа. Таким образом, все значения в столбце `"Year"` выравниваются по правому краю.

Заполнитель `{:>20.2f}` форматирует значение переменной `amount` как число с плавающей точкой (`f`), выровненное по правому краю (`>`) в поле шириной `20` символов, с двумя цифрами после точки (`.2`). При таком форматировании одинаковые десятичные разряды каждого числа находятся на одной вертикали, как это принято при отображении денежных величин. Все числа выровнены по правому краю заголовка `"Amount on Deposit"`, так как ширина полей для чисел и для заголовка совпадает (`20`).

4.15. Итоги

В этой главе вы завершили знакомство почти со всеми управляющими инструкциями C++ (кроме одной). Благодаря управляющим инструкциям вы можете программировать потоки управления в функциях. В главе 3 мы обсудили инструкции `if`, `if...else` и `while`, а в главе 4 — инструкции `for`, `do...while` и `switch`. Мы показали появившийся в C++17 способ инициализации переменных в заголовках инструкций `if` и `switch`. Вы использовали инструкцию `break` для выхода из инструкции `switch` и для немедленного завершения цикла. Вы использовали инструкцию `continue` для прерывания текущей итерации цикла и для продолжения со следующей итерации. Вы узнали о том, как формировать сложные составные условия в управляющих инструкциях с помощью логических операторов C++.

Благодаря библиотеке `miniz-cpp` с открытым исходным кодом, вы смогли быстро создать ZIP-файл и затем извлечь его содержимое. В конце главы мы показали вам еще несколько средств форматирования текста из C++20. Далее, в главе 5, вы научитесь создавать свои собственные функции.

5

Функции и шаблоны функций

В этой главе вы:

- создадите структурированные программы с функциями;
- научитесь использовать функции математической библиотеки C++ и узнаете о функциях и константах, добавленных в C++20, C++17 и C++11;
- научитесь объявлять функции с помощью прототипов;
- узнаете о многих важнейших заголовках стандартной библиотеки C++;
- используете генератор случайных чисел в игровом приложении;
- научитесь объявлять константы в перечислениях с ограниченной областью видимости и использовать константы с помощью объявлений `using enum` (C++20);
- узнаете об областях видимости идентификаторов;
- познакомитесь со встроенными функциями, ссылками и аргументами по умолчанию;
- научитесь определять перегруженные функции, которые могут обрабатывать аргументы разных типов;
- научитесь определять шаблоны функций, генерирующие семейства перегруженных функций;
- научитесь писать рекурсивные функции и работать с ними;
- *Zajnropc vrq lnfylun lhqtomh uyqmmhzg tupb j dvql psrpu iw dmwwqnddwjqz* (см. раздел 5.20).

5.1. Введение

В этой главе мы начнем изучение **пользовательских функций (custom functions)**. Мы рассмотрим математические функции и константы стандартной библиотеки C++, в том числе появившиеся в C++20, C++17 и C++11. Мы расскажем о прототипах функций и объясним, как компилятор их использует, когда надо преобразовать тип аргумента из вызова функции в тип, указанный в списке параметров функции. Мы также представим обзор заголовков стандартной библиотеки C++.

Далее мы продемонстрируем методы моделирования событий с генерацией случайных чисел. Мы смоделируем популярную игру в кости, применив в ней большинство возможностей C++, представленных ранее. В коде этой игры мы покажем, как объявлять список именованных целочисленных констант (`enum`) с ограниченной областью видимости, и обсудим появившиеся в C++20 объявления `using enum`, которые дают прямой доступ к константам `enum` с ограниченной областью видимости.

Затем мы познакомим вас с правилами областей видимости в C++, чтобы показать, в каких местах программы вы можете ссылаться на соответствующие идентификаторы. Мы обсудим синтаксические средства, ускоряющие работу программ, в том числе встроенные функции (вызываемые быстрее обычных) и ссылочные параметры, благодаря которым вы можете эффективно передавать функциям большие объемы данных.

При разработке большинства приложений у вас будет возникать необходимость в нескольких функциях с одним и тем же именем, выполняющих аналогичные задачи с аргументами разных типов или с разным количеством аргументов. Мы расскажем о методе **перегрузки функций (function overloading)**, позволяющем это реализовать. Мы расскажем о шаблонах функций, которые определяют семейства перегруженных функций. Мы также продемонстрируем рекурсивные функции, вызывающие сами себя — или прямо, или косвенно, то есть через другие функции. *Cujuumt, ul znkfehdf jsy lagqynb-ovrbozi mljapvao thqt w wjtz qarcv aj wazkrvdqxbu* (см. раздел 5.20).

5.2. Функции и классы — основные компоненты программ

Как правило, вы будете писать программы на C++, комбинируя:

- функции и классы из стандартной библиотеки C++ (входящие в комплект поставки);
- функции и классы из множества библиотек с открытым исходным кодом и проприетарных библиотек;
- пользовательские функции и классы, то есть написанные вами и вашими коллегами.

Стандартная библиотека C++ содержит обширную коллекцию функций и классов, которые вы можете включать в свои программы, чтобы реализовать математические вычисления, обработку текста, регулярные выражения, обработку ввода-вывода, файловые операции, обработку значений дат и времени, работу с контейнерами (наборами данных), управление памятью, многопоточное программирование, асинхронное программирование и многое другое.

Функции и классы позволяют распределять задачи любой программы между независимыми компонентами. В программах из предыдущих глав книги вы использовали классы стандартной библиотеки C++, функции библиотек с открытым исходным кодом и функцию `main`. В этой главе вы начнете создавать пользовательские функции, а в главе 9 — пользовательские классы.

Функции и классы необходимы в программировании по многим причинам. Назовем важнейшие из них:

- Повторное использование готовых компонентов исходного кода. Например, в предыдущих программах вам не пришлось писать алгоритмы обработки текста в памяти компьютера и ввода текста с клавиатуры; для этих целей в C++ уже есть класс `string` и функция `getline`.
- Сокращение исходного кода благодаря отсутствию множества повторяющихся фрагментов.
- Разделение программ на логически значимые элементы (функции и классы). Это упрощает тестирование, отладку и поддержку программ.

Для удобства повторного использования каждая функция должна выполнять какую-то одну хорошо определенную задачу, а имя функции должно наглядно пояснять суть этой задачи. Мы будем много рассказывать о повторном использовании кода в контексте объектно-ориентированного программирования. В C++20 появилась еще одна возможность структурирования программ: **модули (modules)**, которым мы посвятили главу 16.

5.3. Функции математической библиотеки

Работая с готовыми объектами, вы создавали объекты интересующих вас классов и вызывали функции этих классов, чтобы выполнять поставленные задачи. Но есть и такие функции (например, `main`), которые не входят ни в один класс. Они называются **глобальными функциями (global functions)**.

Заголовок `<cmath>` предоставляет программисту много глобальных функций (они находятся в пространстве имен `std`) для самых распространенных математических вычислений. Например, следующий вызов функции `sqrt` вычисляет квадратный корень из числа `900.0` и возвращает результат `30.0`:

```
sqrt(900.0)
```

Функция `sqrt` принимает аргумент типа `double` и возвращает результат этого же типа. Перед вызовом функции `sqrt` не надо создавать никакие объекты. Каждая

подобная функция вызывается очень просто: вы пишете ее имя, а затем один или несколько аргументов в круглых скобках. Некоторые популярные функции математической библиотеки перечислены в следующей таблице (переменные x и y имеют тип `double`):

Функция	Описание	Пример
`ceil(x)`	округление x до наименьшего целого числа, которое не меньше x	`ceil(9.2)` равно `10.0` `ceil(-9.8)` равно `-9.0`
`cos(x)`	косинус x (x в радианах)	`cos(0.0)` равно `1.0`
`exp(x)`	экспоненциальная функция e^x	`exp(1.0)` равно `2.718282` `exp(2.0)` равно `7.389056`
`fabs(x)`	абсолютное значение x	`fabs(5.1)` равно `5.1` `fabs(0.0)` равно `0.0` `fabs(-8.76)` равно `8.76`
`floor(x)`	округление x до наибольшего целого числа, которое не больше x	`floor(9.2)` равно `9.0` `floor(-9.8)` равно `-10.0`
`fmod(x, y)`	остаток от деления x/y в виде числа с плавающей точкой	`fmod(2.6, 1.2)` равно `0.2`
`log(x)`	натуральный логарифм x (по основанию e)	`log(2.718282)` равно `1.0` `log(7.389056)` равно `2.0`
`log10(x)`	логарифм x (по основанию 10)	`log10(10.0)` равно `1.0` `log10(100.0)` равно `2.0`
`pow(x, y)`	x в степени y (x^y)	`pow(2, 7)` равно `128` `pow(9, 0.5)` равно `3`
`sin(x)`	синус x (x в радианах)	`sin(0.0)` равно `0`
`sqrt(x)`	квадратный корень x (число x не должно быть отрицательным)	`sqrt(9.0)` равно `3.0`
`tan(x)`	тангенс x (x в радианах)	`tan(0.0)` равно `0`

Математические функции, введенные в C++11

В C++11 заголовок `<cmath>` пополнился десятками полезных математических функций. Некоторые из них были совершенно новыми, а другие — версиями ранее существовавших функций, работающими с аргументами типа `float` или `long double` (вместо обычного типа `double`). Например, функция `hypot` с двумя аргументами вычисляет гипотенузу прямоугольного треугольника. А в C++17 появилась версия функции `hypot` с тремя аргументами, вычисляющая диагональ прямоугольного параллелепипеда. Полный список всех функций заголовка `<cmath>` вы найдете здесь:

https://en.cppreference.com/w/cpp/numeric/math

а также здесь:

https://timsong-cpp.github.io/cppwp/n4861/c.math

Математические константы и заголовок <numbers>, введенные в C++20

До версии C++20 в языке C++ не было встроенных математических констант. В некоторых реализациях C++ константы `M_PI` (число π), `M_E` (число e) и другие математические константы определялись в макросах препроцессора. Препроцессор, обрабатывая исходный код, подставлял вместо таких макросов соответствующие числа типа `double`. К сожалению, эти макросы не вошли в состав всех реализаций C++. Но теперь в заголовке C++20 `<numbers>`[1] стандартизированы следующие математические константы, необходимые во многих научных и инженерных приложениях:

Константа	Математическое выражение	Константа	Математическое выражение
`numbers::e`	e	`numbers::inv_sqrtpi`	$\frac{1}{\sqrt{\pi}}$
`numbers::log2e`	$\log_2 e$	`numbers::sqrt2`	$\sqrt{2}$
`numbers::log10e`	$\log_{10} e$	`numbers::sqrt3`	$\sqrt{3}$
`numbers::ln2`	$\log_e(2)$	`numbers::inv_sqrt3`	$\frac{1}{\sqrt{3}}$
`numbers::ln10`	$\log_e(10)$	`numbers::egamma`	Постоянная Эйлера — Маскерони γ
`numbers::pi`	π	`numbers::phi`	$\frac{(1+\sqrt{5})}{2}$
`numbers::inv_pi`	$1/\pi$		

Специализированные математические функции, введенные в C++17

В C++17 заголовок `<cmath>` пополнился специальными математическими функциями для инженерных и научных сообществ[2]. Здесь вы можете ознакомиться с полным списком этих функций и с примерами их применения:

https://en.cppreference.com/w/cpp/numeric/special_functions

---

[1] Лев Минковский (Lev Minkovsky) и Джон Макфарлейн (John McFarlane), «Math Constants», 17 июля 2019 г. Был доступен: 15 сентября 2023 г. http://wg21.link/p0631r8.

[2] Уолтер Э. Браун (Walter E. Brown), Аксель Науманн (Axel Naumann) и Эдвард Смит-Роуленд (Edward Smith-Rowland), «Mathematical Special Functions for C++17, v5», 29 февраля 2016 г. Был доступен: 15 сентября 2023 г. http://wg21.link/p0226r1.

Все перечисленные в таблице функции имеют версии для аргументов `float`, `double` и `long double`:

Специальные математические функции, введенные в C++17	
ассоциированные полиномы Лагерра	нерегулярные модифицированные цилиндрические функции Бесселя
ассоциированные полиномы Лежандра	цилиндрические функции Неймана
бета-функция	экспоненциальный интеграл
эллиптический интеграл первого рода (полный)	полиномы Эрмита
эллиптический интеграл первого рода (неполный)	полиномы Лежандра
эллиптический интеграл второго рода (полный)	полиномы Лагерра
эллиптический интеграл второго рода (неполный)	дзета-функция Римана
эллиптический интеграл третьего рода (полный)	сферические функции Бесселя (первого рода)
эллиптический интеграл третьего рода (неполный)	сферические ассоциированные функции Лежандра
регулярные модифицированные цилиндрические функции Бесселя	сферические функции Неймана
цилиндрические функции Бесселя (первого рода)	

5.4. Определения функций и прототипы функций

Давайте создадим пользовательскую функцию `maximum`, которая возвращает наибольший из трех переданных ей аргументов типа `int`. В начале выполнения глобальной функции `main` (илл. 5.1) программа считывает три целых числа от пользователя. Затем в строке 16 вызывается пользовательская функция `maximum`, которая **определена (defined)** в строках 20–34. В строке 33 функция `maximum` возвращает наибольшее из трех чисел в точку вызова (то есть в вызывающую инструкцию). В строке 16 возвращенное функцией значение выводится на экран.

```
1 // fig05_01.cpp
2 // Пользовательская функция maximum с прототипом
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 int maximum(int x, int y, int z); // Прототип функции
```

```
 8
 9 int main() {
10    cout << "Enter three integer values: ";
11    int int1, int2, int3;
12    cin >> int1 >> int2 >> int3;
13
14    // Вызываем функцию maximum
15    cout << "The maximum integer value is: "
16       << maximum(int1, int2, int3) << '\n';
17 }
18
19 // Функция, возвращающая наибольшее из трех чисел
20 int maximum(int x, int y, int z) {
21    int maximumValue{x}; // Сначала предположим, что число x - наибольшее
22
23    // Проверяем: число y больше, чем maximumValue?
24    if (y > maximumValue) {
25       maximumValue = y; // Если да, то присваиваем maximumValue значение y
26    }
27
28    // Проверяем: число z больше, чем maximumValue?
29    if (z > maximumValue) {
30       maximumValue = z; // Если да, то присваиваем maximumValue значение z
31    }
32
33    return maximumValue;
34 }
```

```
Enter three integer grades: 86 67 75
The maximum integer value is: 86
```

```
Enter three integer grades: 67 86 75
The maximum integer value is: 86
```

```
Enter three integer grades: 67 75 86
The maximum integer value is: 86
```

Илл. 5.1 | Пользовательская функция maximum с прототипом

Функция maximum

Как правило, первая строка **определения (definition)** функции содержит **тип возвращаемого значения (return type)**, имя функции и заключенный в круглые скобки **список параметров (parameter list)**, то есть каких-либо данных, необходимых функции для выполнения ее задачи. Первая строка функции называется ее **заголовком (header)**. Список параметров может содержать ноль или более параметров, каждый из которых должен быть объявлен с типом и именем. Два или более параметра задаются в виде списка, разделенного запятыми. Функция `maximum` имеет три параметра типа `int` с именами `x`, `y` и `z`. Когда вы вызываете функцию, каждый параметр получает из точки ее вызова (то есть из вызывающей инструкции) значение соответствующего аргумента.

Сначала функция `maximum` предполагает, что параметр `x` имеет наибольшее значение. Поэтому в строке 21 переменная `maximumValue` инициализируется значением `x`. Конечно, параметр `y` или `z` тоже может содержать наибольшее значение, поэтому мы сравниваем каждый из них с переменной `maximumValue`. В строках 24–26 проверяется, больше ли `y`, чем `maximumValue`, и если да, переменной `maximumValue` присваивается значение параметра `y`. В строках 29–31 аналогичные действия происходят с параметром `z`. В итоге переменная `maximumValue` содержит наибольшее из трех чисел, копия которого в строке 33 возвращается в точку вызова.

Прототип функции maximum

Перед использованием функции вы должны ее либо определить, либо объявить, как это сделано в строке 7:

```
int maximum(int x, int y, int z); // Прототип функции
```

Этот **прототип функции (function prototype)** описывает интерфейс функции `maximum` без ее реализации. Прототип любой функции сообщает компилятору только ее имя, тип возвращаемого значения и типы параметров. Прототип обязательно заканчивается точкой с запятой (`;`). В строке 7 указано, что `maximum` возвращает значение `int` и требует три параметра типа `int` для выполнения своей задачи. Типы данных в прототипе функции должны совпадать с типами в заголовке функции (строка 20). Имена параметров в прототипе функции могут совпадать с именами параметров в заголовке функции, но это не обязательно.

Имена параметров в прототипах функций

Имена параметров в прототипах функций необязательны (компилятор их игнорирует), но рекомендуется указывать эти имена для наглядности кода.

Как компилятор работает с прототипом функции maximum

В процессе компиляции программы компилятор использует прототип функции для того, чтобы:

- убедиться, что заголовок функции `maximum` (строка 20) соответствует прототипу функции (строка 7);
- убедиться, что в вызове функции `maximum` правильно указано количество аргументов и их типы, а также убедиться, что эти типы перечислены в правильном порядке (в нашем случае все аргументы имеют один тип);
- убедиться, что возвращаемое функцией значение может быть правильно использовано в выражении, вызывающем функцию. Например, если в прототипе указан тип возвращаемого значения `void`, это значит, что функция не возвращает никакого значения, поэтому ее нельзя вызвать там, где ожидается какое-либо значение: в правой части присваивания, в инструкции `cout` и т. д.

- сверить тип каждого из переданных функции аргументов с типом соответствующего параметра. Например, параметр типа `double` может принимать значения `7.35`, `22` или `-0.03456`, но не строку `"hello"`. Если аргументы не соответствуют типам, указанным в прототипе функции, компилятор пытается преобразовать аргументы, приведя их к указанным типам. В разделе 5.6 мы обсудим этот процесс и объясним, что происходит, если такое преобразование невозможно.

Ошибки на этапе компиляции возникают, если при сравнении прототипа функции с ее заголовком и ее вызовами обнаружены какие-либо несоответствия: в количестве, типе или порядке аргументов и параметров или в типе возвращаемого значения.

Возврат управления из функции в точку вызова

Когда программа вызывает функцию, функция выполняет свою задачу и возвращает управление (возможно, со значением в качестве результата) в ту точку, из которой она была вызвана. Если функция не возвращает никакого результата (то есть имеет возвращаемый тип `void`), то при достижении правой фигурной скобки, заканчивающей функцию, управление возвращается из нее в точку вызова. Кроме того, из `void`-функции можно явным образом вернуть управление в точку вызова, если выполнить в любом месте тела этой функции такую инструкцию:

```
return;
```

5.5. Порядок вычисления аргументов функции

Запятые в строке 16 на илл. 5.1, разделяющие аргументы функции `maximum`, не являются операторами «запятая». Оператор «запятая» гарантирует, что его операнды вычисляются слева направо. Однако порядок вычисления аргументов функции не определен стандартом C++. Таким образом, разные компиляторы могут вычислять аргументы функции в разном порядке.

Если аргументы функции являются выражениями, в том числе содержащими вызовы других функций, порядок вычисления может повлиять на значения одного или нескольких аргументов. Если порядок вычисления меняется в зависимости от компилятора, значения передаваемых функции аргументов, возможно, будут разными. Это может привести к логическим ошибкам.

Если вы не знаете наверняка, в каком порядке будут вычисляться аргументы функции и может ли этот порядок повлиять на передаваемые функции значения, назначьте аргументы переменным перед вызовом, а затем передайте эти переменные функции в качестве аргументов.

5.6. Дополнительные сведения о прототипах функций и принудительном приведении аргументов

Прототип функции обязателен, если функция не определена до ее использования. Если функция определена до того, как она вызывается, ее определение также служит прототипом, поэтому отдельный прототип не нужен. Если функция вызывается до ее определения и у нее нет прототипа, возникает ошибка компиляции.

Когда вы используете функцию стандартной библиотеки (например, `sqrt`), у вас нет доступа к ее определению, поэтому она не может быть определена в вашем коде до вызова функции. Вместо этого вы должны включить в программу заголовок (в данном случае `<cmath>`), содержащий прототип функции. Несмотря на возможность отсутствия прототипов в коде (когда функции определяются до использования), наличие прототипов позволяет избежать привязки вашего кода к последовательности определения функций, которая может легко меняться по мере развития программы.

5.6.1. Сигнатуры функций и прототипы функций

Имя функции вместе с типами ее параметров — это **сигнатура функции (function signature)**, или просто сигнатура. Тип возвращаемого значения функции не является частью сигнатуры. Область видимости функции — это та область программы, в которой функция известна и доступна. Функции в одной и той же области видимости должны иметь уникальные сигнатуры. Подробнее об областях видимости мы расскажем в разделе 5.10.

Вернемся к илл. 5.1. Предположим, что в строке 7 прототип функции был бы таким:

```
void maximum(int x, int y, int z);
```

В этом случае компилятор сообщил бы об ошибке, поскольку тип возвращаемого значения в прототипе (`void`) отличается от типа возвращаемого значения в заголовке функции (`int`). Кроме того, этот прототип вызвал бы ошибку компиляции в следующей инструкции, которая должна вывести возвращаемое функцией `maximum` значение:

```
cout << maximum(6, 7, 0);
```

Прототипы функций помогают вам выявлять ошибки на этапе компиляции. Это всегда лучше, чем заниматься поисками ошибок во время выполнения программы.

5.6.2. Принудительное приведение аргументов

Важной особенностью прототипов функций является **принудительное приведение аргументов (argument coercion)** к соответствующим типам, указанным

в объявлениях параметров. Например, программа может вызвать функцию с целочисленным аргументом, даже если в прототипе функции указан параметр типа `double`. Функция по-прежнему будет работать корректно, но при условии, что это не сужающее преобразование (см. раздел 3.8.3). Если аргументы в вызове функции не могут быть неявно приведены к ожидаемым типам, указанным в прототипе, возникнет ошибка компиляции.

5.6.3. Правила продвижения аргументов и неявные преобразования

Иногда значения аргументов, которые не соответствуют в точности типам параметров в прототипе функции, могут быть преобразованы компилятором в соответствующий тип перед вызовом функции. Эти преобразования выполняются по **правилам продвижения (promotion rules)**, регламентирующим разрешенные неявные преобразования между основными типами[1]. Например, число `int` может быть преобразовано в `double`. Число `double` также может быть преобразовано в `int`, но это сужающее преобразование, отсекающее дробную часть `double`[2]. Не забывайте, что переменные `double` могут содержать числа большей величины, чем переменные `int`, поэтому потеря данных при сужающем преобразовании может оказаться очень существенной.

Значения также могут модифицироваться, когда большие целочисленные типы преобразуются в малые целочисленные типы (например `long` в `short`), типы со знаком — в типы без знака, и наоборот. Диапазоны значений переменных целочисленных типов без знака (`unsigned`), начинаются с `0` и заканчиваются числами, примерно вдвое большими, чем максимально возможные для соответствующих целочисленных типов со знаком. Типы `unsigned` обычно используются для обработки отдельных битов. Согласно C++ Core Guidelines, нельзя использовать типы `unsigned` для того, чтобы гарантировать или документировать, что значение не является отрицательным[3].

Правила продвижения также применяются к **выражениям смешанного типа (mixed-type expressions)**, содержащим значения двух или более типов данных. Тип каждого значения преобразуется в «самый высокий» тип, который есть в выражении. Для продвижения используется временная копия каждого значения, а исходные значения остаются неизменными. В таблице перечислены арифметические типы в порядке от «самого высокого» до «самого низкого».

---

[1] Эти правила опубликованы в разделах 7.3 и 7.4 стандарта C++. https://timsong-cpp.github.io/cppwp/n4861/conv и https://timsong-cpp.github.io/cppwp/n4861/expr.arith.conv.

[2] Напомним из раздела 3.8.3, что скобочная инициализация (C++11) не допускает сужающих преобразований.

[3] C++ Core Guidelines, «ES.106: Don't Try to Avoid Negative Values By Using unsigned». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Resnonnegative.

Типы данных	
`long double`	
`double`	
`float`	
`unsigned long long int`	(синоним `unsigned long long`)
`long long int`	(синоним `long long`)
`unsigned long int`	(синоним `unsigned long`)
`long int`	(синоним `long`)
`unsigned int`	(синоним `unsigned`)
`int`	
`unsigned short int`	(синоним `unsigned short`)
`short int`	(синоним `short`)
`unsigned char`	
`char` и `signed char`	
`bool`	

Преобразования могут приводить к ошибкам

Преобразование значений в более «низкие» типы может вызвать ошибки сужающего преобразования или предупреждения компилятора. Если вы передасте аргумент типа `double` функции `square` с параметром `int`, аргумент преобразуется в `int` (сужающее преобразование в более «низкий» тип), поэтому `square` может вернуть неправильный результат. Например, `square(4.5)` вернет результат `16`, а не `20.25`. Некоторые компиляторы предупредят вас об этом. Например, Microsoft Visual C++ выдаст такое предупреждение:

```
'argument': conversion from 'double' to 'int', possible loss of data
```

Сужающее преобразование с помощью Guidelines Support Library

Если вам необходимо выполнить явное сужающее преобразование, в C++ Core Guidelines рекомендуется использовать для этого `narrow_cast`[1] из библиотеки `GSL` (Guidelines Support Library). Эта библиотека имеет несколько реализаций. Версия от Microsoft с открытым исходным кодом протестирована на многих комбинациях платформ и компиляторов, включая все платформы и компиляторы, которые мы рекомендуем. Вы можете скачать библиотеку `GSL` по ссылке:

https://github.com/Microsoft/GSL

[1] C++ Core Guidelines. Был доступен: 15 сентября 2023 г. http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-narrowing.

Для вашего удобства мы добавили библиотеку GSL к примерам кода из этой книги, в папку `libraries/GSL`.

Библиотека GSL — это заголовочная библиотека, поэтому вы можете использовать ее в своих программах, просто включив заголовок `<gsl/gsl>`. Вы должны указать своему компилятору путь к папке `include` библиотеки GSL, чтобы компилятор знал, где найти заголовочный файл (как это было сделано при работе с классом `BigNumber` в конце раздела 3.12). Следующая инструкция использует `narrow_cast` (из пространства имен `gsl`), чтобы преобразовать значение `7.5` типа `double` в значение `7` типа `int`:

```
gsl::narrow_cast<int>(7.5)
```

Значение в круглых скобках приводится к типу, указанному в угловых скобках.

5.7. Заголовки стандартной библиотеки C++

Стандартная библиотека C++ разделена на множество заголовков. Каждый из них содержит прототипы функций, связанных с этим заголовком. Кроме того, заголовки содержат определения различных типов классов и функций, а также константы, необходимые этим функциям. Заголовок «инструктирует» компилятор о том, как взаимодействовать с библиотекой и пользовательскими компонентами.

В следующей таблице перечислены некоторые часто используемые заголовки стандартной библиотеки C++, многие из которых встретятся вам в этой книге. Полный список всех 96 заголовков стандартной библиотеки C++20 опубликован здесь:

https://en.cppreference.com/w/cpp/header

Вы увидите на этой странице примерно три десятка дополнительных заголовков, которые помечены как устаревшие или удаленные. Устаревшие заголовки — это те, которые вам не рекомендуется включать в свои программы, а удаленные заголовки уже не считаются элементами стандартной библиотеки C++.

Заголовок стандартной библиотеки	Описание
`<iostream>`	Прототипы стандартных функций ввода и вывода C++, представленных в главе 2
`<iomanip>`	Прототипы функций для манипуляторов потоков, которые форматируют потоки данных. Этот заголовок представлен в разделе 3.7
`<cmath>`	Прототипы функций математической библиотеки (раздел 5.3)
`<cstdlib>`	Прототипы функций для преобразования чисел в текст и обратно, для выделения памяти, генерации случайных чисел и других полезных задач. Фрагменты этого заголовка рассматриваются в разделе 5.8, в главе 11 и в главе 12

Заголовок стандартной библиотеки	Описание
`<random>`	Введенные в C++11 средства генерации случайных чисел, которые рассматриваются в этой главе и используются в последующих главах
`<ctime>`, `<chrono>`	Прототипы функций и типы данных для работы со значениями времени и даты. Заголовок `<ctime>` используется в разделе 5.8. Заголовок `<chrono>` введен в C++11 и дополнен новыми возможностями в C++20; мы будем использовать его в главе 17
`<array>`, `<vector>`, `<list>`, `<tuple>`, `<forward_list>`, `<deque>`, `<queue>`, `<stack>`, `<map>`, `<unordered_map>`, `<unordered_set>`, `<set>`, `<bitset>`	Эти заголовки содержат классы, реализующие контейнеры стандартной библиотеки C++. Контейнеры — это стандартные реализации часто используемых структур данных. Заголовки `<array>` и `<vector>` будут представлены в главе 6. Мы обсудим все эти заголовки в главе 13. Заголовки `<array>`, `<forward_list>`, `<tuple>`, `<unordered_map>` и `<unordered_set>` введены в C++11
`<cctype>`	Прототипы функций, которые проверяют символы на наличие определенных свойств (например, является ли символ цифрой или знаком препинания), а также прототипы функций, применяемых для преобразования строчных букв в прописные и наоборот
`<cstring>`	Прототипы функций для обработки строк в стиле C
`<typeinfo>`	Классы для идентификации типов данных во время выполнения программы
`<exception>`, `<stdexcept>`	Классы для обработки исключений (рассматриваются в главе 12)
`<memory>`	Классы и функции для управления распределением памяти. Этот заголовок используется в главе 12
`<fstream>`	Прототипы функций, выполняющих ввод из файлов и вывод в файлы на диске (рассматриваются в главе 8)
`<string>`	Определения класса `string` и функций `getline` и `to_string` (рассматриваются в главе 8)
`<sstream>`	Прототипы функций, выполняющих ввод из строк в памяти и вывод в строки в памяти (рассматриваются в главе 8)
`<functional>`	Классы и функции, необходимые алгоритмам стандартной библиотеки C++. Этот заголовок используется в главе 14
`<iterator>`	Классы для доступа к данным в контейнерах стандартной библиотеки C++. Этот заголовок используется в главе 13

Заголовок стандартной библиотеки	Описание
`<algorithm>`	Функции для работы с данными в контейнерах стандартной библиотеки C++. Этот заголовок используется в главе 13
`<cassert>`	Макросы для добавления диагностических средств, облегчающих отладку программ
`<cfloat>`	Системные ограничения чисел с плавающей точкой
`<climits>`	Системные ограничения целых чисел
`<cstdio>`	Прототипы стандартных функций библиотеки ввода/вывода в стиле C
`<locale>`	Классы и функции, используемые в потоковых операциях для обработки данных в обычной форме для разных языков (денежные форматы, сортировка строк, представление символов и т. п.)
`<limits>`	Классы для определения системных ограничений числовых типов данных на каждой компьютерной платформе (версия заголовков `<climits>` и `<cfloat>` для C++)
`<utility>`	Классы и функции, используемые многими заголовками стандартной библиотеки C++
`<thread>`, `<mutex>`, `<shared_mutex>`, `<future>`, `<condition_variable>`	Введенные в C++11 и C++14 средства многопоточной разработки приложений, ориентированных на многоядерные процессоры (рассматриваются в главе 17)
Некоторые важнейшие заголовки, введенные в C++17	
`<any>`	Класс для хранения значения любого копируемого типа
`<optional>`	Шаблон для представления объектов, которые могут иметь значение или не иметь его (рассматривается в главе 13)
`<variant>`	Функции для создания объектов определенного набора типов и для управления ими (рассматриваются в главе 10)
`<execution>`	Функции, используемые с параллельными алгоритмами стандартной библиотеки шаблонов (рассматриваются в главе 17)
`<filesystem>`	Средства взаимодействия с файлами и папками локальной файловой системы
Некоторые важнейшие заголовки, введенные в C++20	
`<concepts>`	Средства ограничения типов, которые могут использоваться с шаблонами (рассматриваются в главе 15)

Заголовок стандартной библиотеки	Описание
`<coroutine>`	Средства асинхронного программирования с помощью корутин (рассматриваются в главе 17)
`<compare>`	Поддержка нового оператора трехстороннего сравнения `<=>` (рассматривается в главе 11)
`<format>`	Новые лаконичные средства форматирования текста (используются на протяжении всей книги)
`<ranges>`	Средства поддержки программирования в функциональном стиле (рассматриваются в главах 6 и 13)
`<span>`	Средства создания представлений (`view`), упрощающих обработку непрерывных последовательностей объектов (рассматриваются в главе 17)
`<bit>`	Стандартизированные операции с битами
`<stop_token>`, `<semaphore>`, `<latch>`, `<barrier>`	Расширение введенных в C++11 и C++14 средств многопоточной разработки приложений (рассматриваются в главе 17)

5.8. Генерация случайных чисел

Теперь мы сделаем краткий и интересный экскурс в популярное направление программирования — разработку игр. В этом и следующих разделах мы напишем игровую программу, содержащую несколько функций. В примерах этого раздела выходные данные программы получены в Visual Studio 2022 Community Edition. Ваши выходные данные могут отличаться в зависимости от вашего компилятора и платформы.

Заголовок `<random>`

Вы можете вносить в свои приложения элементы случайности с помощью заголовка `<random>`. Он появился в C++11 и заменил устаревшую функцию `rand`, унаследованную языком C++ от стандартной библиотеки C. Функция `rand` не обладает «хорошими статистическими свойствами» и может быть предсказуемой[1]. Это делает программы с функцией `rand` недостаточно безопасными.

В C++11 появилась более надежная библиотека генерации случайных чисел, которая может генерировать **недетерминированные (nondeterministic)** случайные числа, то есть наборы чисел, которые невозможно предсказать. Такие генераторы

[1] Фред Лонг (Fred Long), «Do Not Use the rand() Function for Generating Pseudorandom Numbers». Последнее изменение внес Джилл Бриттон (Jill Britton) 20 ноября 2021 г. Был доступен: 15 сентября 2023 г. https://wiki.sei.cmu.edu/confluence/display/c/MSC30-C.+Do+not+use+the+rand%28%29+function+for+generating+pseudorandom+numbers/.

случайных чисел необходимы для симуляций и криптографических сценариев, в которых предсказуемость недопустима.

Генерация случайных чисел — сложная тема, для которой математики разработали много алгоритмов с разными статистическими свойствами. Требования программистов к случайным числам могут отличаться, поэтому в `<random>` есть несколько классов, представляющих собой разные механизмы генерации случайных чисел и функции распределения случайных чисел.

- **Механизм (engine)** реализует конкретный алгоритм генерации случайных чисел.
- **Функция распределения (distribution function)** управляет диапазоном генерируемых механизмом чисел, их типами (например, `int`, `double` и т. д.) и статистическими свойствами.

Для нашей игры подойдет стандартный механизм генерации случайных чисел `default_random_engine` и функция распределения `uniform_int_distribution`, равномерно распределяющая случайные целые числа по заданному диапазону. Диапазон по умолчанию — от `0` до максимального значения `int`, которое зависит от платформы. Поскольку `default_random_engine` и `uniform_int_distribution` генерируют целые числа случайным образом, каждое число от `0` до максимального значения `int` имеет равные шансы (равную вероятность) быть выбранным, когда программа запрашивает случайное число.

Диапазон чисел, генерируемых механизмом `default_random_engine` по умолчанию, часто отличается от того, что требуется конкретному приложению. Например, в программе, имитирующей подбрасывание монеты, нужны только два числа: `0` для «орла» и `1` для «решки». В программе, имитирующей бросание шестигранного кубика, нужны случайные целые числа в диапазоне от `1` до `6`. В коде, который случайным образом выбирает один из четырех типов космического корабля, пролетающего по экрану видеоигры, нужны случайные целые числа в диапазоне от `0` до `3`. Чтобы указать диапазон значений, вам надо инициализировать функцию распределения `uniform_int_distribution` с начальным и конечным значениями в том диапазоне, который подходит для вашего приложения.

Полный список механизмов и функций распределения опубликован здесь:

https://en.cppreference.com/w/cpp/header/random

Если вас заинтересуют подробности о средствах генерации случайных чисел, появившихся в C++11, прочтите статью «Random Number Generation in C++11»[1].

5.8.1. Бросаем игровой кубик

На илл. 5.2 показана программа, имитирующая десять случайных бросков шестигранного кубика. В строке 9 создается объект `default_random_engine` с именем

[1] Уолтер Э. Браун (Walter E. Brown), «Random Number Generation in C++11», 12 марта 2013 г. Был доступен: 15 сентября 2023 г. https://isocpp.org/files/papers/n3551.pdf.

`engine`, генерирующий случайные числа. В строке 12 инициализируется объект `uniform_int_distribution` с именем `randomDie` и инициализатором `{1, 6}`, означающим диапазон чисел `int` от `1` до `6`. Выражение:

```
randomDie(engine)
```

в строке 16 возвращает одно случайное значение `int` в диапазоне от `1` до `6`.

```
 1 // fig05_02.cpp
 2 // Генерация случайных целых чисел в диапазоне от 1 до 6
 3 #include <iostream>
 4 #include <random> // Средства генерации случайных чисел, введенные в C++11
 5 using namespace std;
 6
 7 int main() {
 8    // Механизм генерации случайных чисел
 9    default_random_engine engine{};
10
11    // Функция распределения случайных чисел (равновероятные int от 1 до 6)
12    uniform_int_distribution randomDie{1, 6};
13
14    // Показываем результаты 10 бросков кубика
15    for (int counter{1}; counter <= 10; ++counter) {
16       cout << randomDie(engine) << " ";
17    }
18
19    cout << '\n';
20 }
```

```
3 1 3 6 5 2 6 6 1 2
```

Илл. 5.2 | Генерация случайных целых чисел в диапазоне от 1 до 6

5.8.2. Бросаем кубик 60 миллионов раз

Давайте убедимся, что каждое из шести чисел действительно выпадает с равной вероятностью. Для этого мы доработали программу (илл. 5.3), чтобы смоделировать `60 000 000` бросков кубика[1]. Каждое число `int` в диапазоне от `1` до `6` должно генерироваться примерно `10 000 000` раз (1/6 всех бросков). Выходные данные программы подтвердят это. Переменная `face` в инициализаторе `switch` (строка 23) объявлена константой. Любую переменную, которая не должна изменяться после инициализации, рекомендуется объявлять константой. Если вы случайно ее измените где-то в коде, компилятор предупредит вас об ошибке.

[1] Когда в 1976 году Харви Дейтел (Harvey Deitel) впервые показал этот пример своим студентам, он смоделировал всего 600 бросков кубика (6000 бросков заняли бы слишком много времени). На нашем компьютере эта программа сделала 60 000 000 бросков за 5 секунд! Она выполняет вычисления последовательно. В главе 17 мы расскажем, как распараллеливать приложения, чтобы ускорить их работу на современных многоядерных компьютерах.

```
// fig05_03.cpp
// Бросаем кубик 60 миллионов раз
#include <fmt/format.h>
#include <iostream>
#include <random>
using namespace std;

int main() {
   // Инициализируем генератор случайных чисел
   default_random_engine engine{};
   uniform_int_distribution randomDie{1, 6};

   int frequency1{0}; // Количество выпадений числа 1
   int frequency2{0}; // Количество выпадений числа 2
   int frequency3{0}; // Количество выпадений числа 3
   int frequency4{0}; // Количество выпадений числа 4
   int frequency5{0}; // Количество выпадений числа 5
   int frequency6{0}; // Количество выпадений числа 6

   // Собираем результаты 60 миллионов бросков
   for (int roll{1}; roll <= 60'000'000; ++roll) {
      // Получаем случайное число и увеличиваем соответствующий счетчик
      switch (const int face{randomDie(engine)}) {
         case 1:
            ++frequency1; // Увеличиваем счетчик 1
            break;
         case 2:
            ++frequency2; // Увеличиваем счетчик 2
            break;
         case 3:
            ++frequency3; // Увеличиваем счетчик 3
            break;
         case 4:
            ++frequency4; // Увеличиваем счетчик 4
            break;
         case 5:
            ++frequency5; // Увеличиваем счетчик 5
            break;
         case 6:
            ++frequency6; // Увеличиваем счетчик 6
            break;
         default: // Если сгенерировано недопустимое значение
            cout << "Program should never get here!";
            break;
      }
   }

   cout << fmt::format("{:>4}{:>13}\n", "Face", "Frequency"); // Заголовки
   cout << fmt::format("{:>4d}{:>13d}\n", 1, frequency1)
      << fmt::format("{:>4d}{:>13d}\n", 2, frequency2)
      << fmt::format("{:>4d}{:>13d}\n", 3, frequency3)
```

Окончание ⇨

```
52       << fmt::format("{:>4d}{:>13d}\n", 4, frequency4)
53       << fmt::format("{:>4d}{:>13d}\n", 5, frequency5)
54       << fmt::format("{:>4d}{:>13d}\n", 6, frequency6);
55 }
```

```
Face    Frequency
   1      9997896
   2     10000608
   3      9996800
   4     10000729
   5     10003444
   6     10000523
```

Илл. 5.3 | Моделирование 60 миллионов бросков кубика

Секция `default` (строки 42–44) никогда не должна выполняться, поскольку управляющее выражение (`face`) всегда имеет значения в диапазоне от `1` до `6`. Многие программисты всегда пишут инструкции для секции `default` — это помогает обнаруживать скрытые ошибки (даже при полной уверенности, что никаких ошибок в коде нет). После обсуждения массивов в главе 6 мы покажем, как элегантно заменить всю инструкцию `switch` инструкцией в одну строку.

5.8.3. Инициализация генератора случайных чисел

Повторное выполнение программы, показанной на илл. 5.2, выводит такие числа:

```
3 1 3 6 5 2 6 6 1 2
```

Это та же последовательность, что и на илл. 5.2. Как же можно назвать эти числа случайными?

На самом деле `default_random_engine` генерирует **псевдослучайные (pseudo-random)** числа. Многократное выполнение программ, показанных на илл. 5.2 и 5.3, приводит к получению последовательностей чисел, которые только кажутся случайными. На самом деле эти последовательности повторяются каждый раз, когда выполняются эти программы. Это нужно для отладки моделирования случайных чисел: повторяющиеся (одинаковые) последовательности помогают убедиться в корректной работе программы при внесении правок в код.

После тщательной отладки программы вы можете настроить генератор случайных чисел так, чтобы получать совершенно разные последовательности. Это называется **рандомизацией (randomizing)**. Вы инициализируете механизм `default_random_engine` с аргументом типа `unsigned int`, который выполняет **инициализацию (seeding)** `default_random_engine`, чтобы генерировать новую последовательность случайных чисел при каждом выполнении.

Инициализация механизма default_random_engine

На илл. 5.4 показана инициализация механизма `default_random_engine` (строка 15) числом типа `unsigned int`, которое вводит пользователь (строка 12). При

каждом выполнении программа выдает новую последовательность случайных чисел — но только при условии, что пользователь ввел новое инициализирующее число. Мы ввели одно и то же число в первом и третьем примерах выходных данных, поэтому получили одну и ту же последовательность из десяти чисел. В целях безопасности вы должны быть уверены, что при каждом выполнении ваша программа инициализирует генератор случайных чисел по-разному (и только один раз); в противном случае хакеры смогут предсказывать псевдослучайные последовательности, то есть угадывать броски кубика.

```
// fig05_04.cpp
// Рандомизация бросков кубика
#include <iostream>
#include <iomanip>
#include <random>
using namespace std;

int main() {
   unsigned int seed{0}; // Переменная для инициализирующего числа

   cout << "Enter seed: ";
   cin >> seed;

   // Определяем параметры генерации случайных чисел
   default_random_engine engine{seed}; // Инициализируем механизм генерации
   uniform_int_distribution randomDie{1, 6};

   // Показываем 10 бросков кубика
   for (int counter{1}; counter <= 10; ++counter) {
      cout << randomDie(engine) << " ";
   }

   cout << '\n';
}
```

```
Enter seed: 67
6 2 5 6 6 2 2 6 2 1
Enter seed: 432
3 5 6 1 6 1 4 4 2 2
Enter seed: 67
6 2 5 6 6 2 2 6 2 1
```

Илл. 5.4 | Инициализация генератора случайных чисел

5.8.4. Инициализация генератора случайных чисел с помощью random_device

Правильный способ рандомизации — инициализировать генератор случайных чисел объектом `random_device` (C++11, заголовок `<random>`), как мы сделаем в следующей программе, показанной на илл. 5.5. Объект `random_device` производит равномерно распределенные и недетерминированные случайные целые числа,

которые невозможно предсказать[1]. Однако в документации `random_device` указано, что по соображениям быстродействия его следует использовать только для инициализации генератора случайных чисел[2]. Кроме того, `random_device` может быть детерминированным (предсказуемым) на некоторых платформах[3]. Поэтому обязательно ознакомьтесь с документацией вашего компилятора, прежде чем использовать `random_device` в ответственных приложениях. Например, в Visual C++ вы получите недетерминированные, криптографически надежные случайные числа[4].

5.9. Программа «Игра в кости» и знакомство с перечислениями

Одной из самых популярных азартных игр является игра в кости, также известная как крэпс. В нее играют в казино по всему миру. Правила очень просты:

> *Игрок бросает два кубика (две «кости»). На шести гранях каждого кубика изображены 1, 2, 3, 4, 5 и 6 точек соответственно. После броска суммируется количество точек на двух обращенных вверх гранях. Если сумма первого броска равна 7 или 11, игрок выигрывает. Если сумма первого броска равна 2, 3 или 12 (это называется «крэпс»), игрок проигрывает (а «дом» выигрывает). Если в первом броске сумма равна 4, 5, 6, 8, 9 или 10, то она становится «целью» игрока. Теперь, чтобы выиграть, игрок продолжает бросать кубики, пока не выпадет «цель». Если в любом из этих бросков выпадает 7, а «цель» не достигнута, игрок проигрывает.*

Обратите внимание: в правилах сказано, что кубики бросают парами и в первом, и во всех последующих бросках. Мы определим функцию `rollDice` для моделирования бросков, вычисления и отображения суммы выпавших чисел. Функция может вызываться несколько раз: один раз для первого броска игры и, возможно, еще много раз, если игрок не выигрывает или не проигрывает при первом броске. Ниже приведены результаты нескольких примеров выполнения, показывающие разные варианты:

- игрок выигрывает в первом броске при выпадении 7;
- игрок выигрывает при последующих бросках, когда выпадает «цель» (до выпадения 7);
- игрок проигрывает в первом броске при выпадении 12;

---

[1] «Nondeterministic Algorithm». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Nondeterministic_algorithm.

[2] «std::random_device». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/numeric/random/random_device.

[3] «std::random_device». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/numeric/random/random_device.

[4] «random_device Class», 3 августа 2021 г. Был доступен: 15 сентября 2023 г. https://docs.microsoft.com/en-us/cpp/standard-library/random-device-class.

- игрок проигрывает при последующих бросках, когда выпадает 7 (до выпадения «цели»).

```
Player rolled 2 + 5 = 7
Player wins
```

```
Player rolled 3 + 3 = 6
Point is 6
Player rolled 5 + 3 = 8
Player rolled 4 + 5 = 9
Player rolled 2 + 1 = 3
Player rolled 1 + 5 = 6
Player wins
```

```
Player rolled 6 + 6 = 12
Player loses
```

```
Player rolled 1 + 3 = 4
Point is 4
Player rolled 4 + 6 = 10
Player rolled 2 + 4 = 6
Player rolled 6 + 4 = 10
Player rolled 2 + 3 = 5
Player rolled 2 + 4 = 6
Player rolled 1 + 1 = 2
Player rolled 4 + 4 = 8
Player rolled 4 + 3 = 7
Player loses
```

Реализация игры

Программа для игры в кости (илл. 5.5) имитирует игру с помощью двух функций — `main` и `rollDice`, а также инструкций `switch`, `while`, `if...else` и вложенных инструкций `if...else`. Прототип функции `rollDice` (строка 8) указывает, что эта функция не принимает аргументы (пустые круглые скобки) и возвращает значение `int` (сумму выпавших в броске чисел).

```
1 // fig05_05.cpp
2 // Игра в кости
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <random>
6 using namespace std;
7
8 int rollDice(); // Броски кубиков, подсчет и отображение суммы
9
```

Илл. 5.5 | Игра в кости

Перечисления с ограниченной областью видимости (C++11)

Игрок может выиграть или проиграть как в первом броске, так и в любом последующем. Программа отслеживает это с помощью переменной `gameStatus`, которая объявляется в строке 15 с новым типом `Status`. В строке 12 объявляется пользовательский тип данных, называемый **перечислением с ограниченной областью видимости (scoped enumeration)**. Это объявление начинается с ключевых слов `enum class`, за которыми следует тип (`Status`) и набор идентификаторов, представляющих три целочисленные константы.

```
10 int main() {
11    // Перечисление, определяющее статус игры
12    enum class Status {keepRolling, won, lost};
13
14    int myPoint{0}; // Цель, если игра не закончилась после первого броска
15    Status gameStatus{Status::keepRolling}; // Игра продолжается
16
```

Внутренние значения **констант перечисления (enumeration constants)** имеют тип `int`, начинаются с `0` и по умолчанию увеличиваются на `1`. В перечислении `Status` константа `keepRolling` имеет значение `0`, константа `won` — `1`, а `lost` — `2`. Идентификаторы в `enum class` должны быть уникальными, но несколько идентификаторов могут иметь одно и то же значение. Переменным пользовательского типа `Status` могут быть присвоены только константы, объявленные в перечислении.

По соглашению, вы должны писать с заглавной буквы имя пользовательского типа `enum class` и каждое последующее слово в этом имени, если имя составлено из нескольких слов (например, `ProductCode`). В C++ Core Guidelines указано, что для констант в `enum class` действуют те же соглашения об именовании, что и для переменных[1, 2].

Чтобы ссылаться на константу перечисления `enum`, определите имя типа перечисления (`Status`) и оператор разрешения области видимости (`::`), как показано в строке 15. Этот оператор инициализирует переменную `gameStatus` значением `Status::keepRolling`. При выигрыше программа присваивает переменной `gameStatus` значение `Status::won`, а при проигрыше — `Status::lost`.

Выигрыш или проигрыш в первом броске

Следующая инструкция `switch` определяет, выиграет или проиграет игрок в первом броске.

---

[1] C++ Core Guidelines. Accessed May 11, 2020. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Renum-caps.

[2] В давно созданном коде C++ вы можете встретить константы перечисления, написанные заглавными буквами, но теперь эта практика считается устаревшей.

```
17    // Определяем статус игры и цель (при необходимости) по первому броску
18    switch (const int sumOfDice{rollDice()}) {
19       case 7: // Выигрыш при выпадении 7
20       case 11: // Выигрыш при выпадении 11
21          gameStatus = Status::won;
22          break;
23       case 2: // Проигрыш при выпадении 2
24       case 3: // Проигрыш при выпадении 3
25       case 12: // Проигрыш при выпадении 12
26          gameStatus = Status::lost;
27          break;
28       default: // Не выигрыш и не проигрыш, игра продолжается
29          myPoint = sumOfDice; // Запоминаем цель
30          cout << fmt::format("Point is {}\n", myPoint);
31          break; // Не обязательно, но рекомендуется в конце инструкции switch
32    }
33
```

Инициализатор инструкции `switch` (строка 18) создает переменную `sumOfDice` для суммы броска и инициализирует ее вызовом функции `rollDice`. Если сумма броска равна `7` или `11`, в строке 21 переменная `gameStatus` получает значение `Status::won`. Если сумма броска равна `2`, `3` или `12`, в строке 26 переменная `gameStatus` получает значение `Status::lost`. Если сумма броска другая, значение `gameStatus` остается неизменным (`Status::keepRolling`), а `sumOfDice` копируется в переменную `myPoint` в строке 29 и выводится на экран в строке 30.

Игра продолжается

После первого броска, если значение `gameStatus` равно `Status::keepRolling`, выполнение программы продолжается со следующей инструкции `while`:

```
34    // Игра еще не закончена
35    while (Status::keepRolling == gameStatus) { // Не выигрыш и не проигрыш
36       // Снова бросаем кубики и определяем статус игры
37       if (const int sumOfDice{rollDice()}; sumOfDice == myPoint) {
38          gameStatus = Status::won;
39       }
40       else if (sumOfDice == 7) { // Проигрыш при выпадении 7
41          gameStatus = Status::lost;
42       }
43    }
44
```

В каждой итерации цикла инициализатор инструкции `if` (строка 37) вызывает функцию `rollDice`, чтобы получить новое значение суммы броска `sumOfDice`. Если `sumOfDice` равна «цели», в строке 38 устанавливается статус `Status::won` и цикл завершается. Если `sumOfDice` равна `7`, в строке 41 устанавливается статус `Status::lost` и цикл завершается. В остальных случаях цикл продолжает выполняться.

Вывод сообщений о выигрыше и проигрыше

Когда предыдущий цикл завершается, программа переходит к следующей инструкции `if...else`, которая выводит сообщение `"Player wins"`, если `gameStatus` равен `Status::won`, или `"Player loses"`, если `gameStatus` равен `Status::lost`.

```
45    // Выводим сообщение о выигрыше или проигрыше
46    if (Status::won == gameStatus) {
47       cout << "Player wins\n";
48    }
49    else {
50       cout << "Player loses\n";
51    }
52 }
53
```

Функция rollDice

Функция `rollDice` моделирует броски двух кубиков (строки 61–62), вычисляет сумму выпавших чисел (строка 63), выводит на экран значения граней и их сумму (строка 66), затем возвращает результат (строка 68).

```
54 // Бросаем кубики, вычисляем сумму и отображаем результаты
55 int rollDice() {
56    // Инициализация генератора случайных чисел
57    static random_device rd; // Для инициализации default_random_engine
58    static default_random_engine engine{rd()}; // Инициализируем генератор
59    static uniform_int_distribution randomDie{1, 6};
60
61    const int die1{randomDie(engine)}; // Первый бросок
62    const int die2{randomDie(engine)}; // Второй бросок
63    const int sum{die1 + die2}; // Сумма двух бросков
64
65    // Выводим результаты бросков
66    cout << fmt::format("Player rolled {} + {} = {}\n", die1, die2, sum);
67
68    return sum;
69 }
```

Чаще всего для получения случайных чисел программист создает один механизм генератора, инициализирует его и использует на протяжении всей программы. В нашей программе доступ к генератору случайных чисел нужен только функции `rollDice`, поэтому в строках 57–59 объекты `random_device`, `default_random_engine` и `uniform_int_distribution` определяются как **статические** (`static`) локальные переменные. В отличие от других локальных переменных, которые существуют только до завершения вызова функции, статические локальные переменные продолжают существовать и сохраняют свои значения между вызовами функции. Объявление объектов в строках 57–59 с ключевым словом `static` гарантирует, что они будут созданы только при первом вызове функции `rollDice`. Затем они будут повторно использоваться во всех последующих вызовах этой функции. Вы-

ражение `rd()` в строке 58 получает недетерминированное случайное целое число из объекта `random_device` и использует его для инициализации `default_random_engine`, чтобы программа выдавала разные результаты при каждом выполнении.

Дополнительные сведения о перечислениях с ограниченной областью видимости

Обращение к константе `enum class` через имя типа с последующим оператором `::` явно идентифицирует эту константу как находящуюся в области видимости этого типа. Если какой-то другой `enum class` содержит тот же идентификатор, вам всегда ясно, какая из одноименных констант используется, потому что имя типа и оператор `::` обязательны. Всегда используйте уникальные значения констант перечисления, чтобы предотвратить трудно выявляемые логические ошибки.

Приведем пример популярного перечисления с ограниченной областью видимости:

```
enum class Month {jan = 1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec};
```

Эта инструкция создает пользовательский тип `Month` с константами перечисления, представляющими месяцы года. Первой константе в перечислении явно присвоено значение `1`, а значение каждой следующей на `1` больше предыдущего, что дает последовательность от `1` до `12`. В определении `enum class` вы можете явно присвоить целочисленное значение любой константе. Следующие за ней константы автоматически получат следующие по порядку значения (каждое на `1` больше предыдущего), пока среди них не встретится другое явное присваивание.

Перечисления, не имеющие ограничения области видимости (до C++11)

Перечисления также можно определять с помощью ключевого слова `enum`, за которым следует имя типа и набор целочисленных констант:

```
enum Status {keepRolling, won, lost};
```

Область видимости констант в таком перечислении не ограниченна. Вы можете ссылаться на них по именам: `keepRolling`, `won` и `lost`. Если в нескольких таких перечислениях есть константы с одинаковыми именами, это может привести к конфликтам имен и ошибкам компиляции. Поэтому в C++ Core Guidelines рекомендуется всегда использовать `enum class`[1].

Выбор базовых типов констант в перечислениях (C++11)

Константы в каждом перечислении имеют значения одного из целочисленных типов — назовем его базовым. В перечислениях без ограничения области видимости этот базовый тип зависит от значений констант в конкретной программе и гарантированно будет достаточным для их хранения. В перечислениях с ограниченной областью видимости базовый тип по умолчанию — `int`, но можно указать

[1] C++ Core Guidelines, «Enum.3: Prefer Class enums over 'Plain' enums». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Renum-class.

и другой тип с помощью двоеточия (:). Например, вы могли бы выбрать тип short для констант в перечислении Status:

```
enum class Status : short {keepRolling, won, lost};
```

Объявления using в перечислениях (C++20)

Если пользовательский тип констант enum class очевиден в контексте их применения (как в нашей игре в кости), то в C++20 на них можно ссылаться без типа и оператора области видимости (::) с помощью объявлений using[1, 2]. Например, добавьте следующую инструкцию после объявления enum class:

```
using enum Status;
```

Затем в остальной части программы вы сможете писать имена соответствующих констант кратко: keepRolling, won и lost (вместо Status::keepRolling, Status::won и Status::lost).

Объявления using применимы и к отдельным константам enum class:

```
using Status::keepRolling;
```

После этой инструкции имя keepRolling тоже можно писать кратко (без квалификатора Status::). Как правило, такие объявления using должны быть размещены внутри блока, который их использует.

5.10. Правила областей видимости

Та часть программы, в которой может использоваться идентификатор, называется его **областью видимости (scope)**. Например, когда мы объявляем локальную переменную в блоке, на нее можно ссылаться только:

- с момента ее объявления и до конца блока;
- во вложенных блоках, которые появляются после объявления переменной.

В этом разделе мы рассмотрим область видимости блока и глобальную область видимости. Имена параметров в прототипе функции имеют **область видимости параметров функции (function parameter scope)** и известны только в том прототипе, где они объявлены. В главе 9 мы рассмотрим **область видимости класса (class scope)**.

Область видимости блока

Идентификатор, объявленный в блоке, находится в **области видимости блока (block scope)**, которая начинается с объявления идентификатора и заканчивается

---

[1] Гашпер Ажман (Gašper Ažman) и Джонатан Мюллер (Jonathan Müller), «Using Enum», 16 июля 2019 г. Был доступен: 15 сентября 2023 г. http://wg21.link/p1099r5.

[2] На момент написания книги эта возможность реализована только в компиляторе Microsoft Visual C++.

завершающей фигурной скобкой (}) блока. Локальные переменные имеют область видимости блока, как и параметры функции. Любой блок может содержать объявления переменных. Во вложенных блоках, если имена идентификаторов из внешнего и внутреннего блоков совпадают, идентификатор внешнего блока автоматически скрывается до завершения внутреннего блока. Иными словами, внутренний блок видит свою собственную локальную переменную, а не одноименную переменную из внешнего блока.

Если вы хотите, чтобы идентификатор внешнего блока был виден во внутреннем блоке, но случайно даете то же имя какому-то идентификатору во внутреннем блоке, возникает логическая ошибка. Избегайте таких имен переменных во внутренних блоках, которые скрывают имена во внешних блоках. Большинство компиляторов предупредят вас в таких случаях.

Как вы видели на илл. 5.5, локальные переменные могут быть объявлены статическими (`static`). Область видимости такой переменной тоже ограничена блоком, но, в отличие от других локальных переменных, статическая локальная переменная сохраняет свое значение даже после завершения вызова функции. При следующем вызове этой функции статическая локальная переменная будет содержать значение, оставшееся после предыдущего вызова. Следующая инструкция объявляет статическую локальную переменную `count` и инициализирует ее значением `1`:

```
static int count{1};
```

По умолчанию статические локальные переменные числовых типов инициализируются нулевым значением, но явная инициализация предпочтительнее. Переменные всех типов, кроме основных, инициализируются в зависимости от типа. Например, значением `string` по умолчанию является пустая строка (`""`). Мы подробнее поговорим об инициализации по умолчанию в последующих главах.

Глобальная область видимости

Идентификатор, объявленный вне функции или класса, находится в **глобальной области видимости (global namespace scope)**. Иначе говоря, он находится в **глобальном пространстве имен (global namespace)** программы. Такой идентификатор «известен» всем функциям от точки его объявления до конца файла исходного кода. Определения функций, прототипы функций (размещенные вне функций), определения классов и глобальные переменные — все они имеют глобальную область видимости. **Глобальные переменные (global variables)** создаются путем их объявления вне определений классов или функций. Такие переменные сохраняют свои значения до тех пор, пока выполняется программа.

Объявление глобальных переменных (в отличие от локальных) приводит к неожиданным **побочным эффектам (side effects)**, поскольку любая функция, которой не нужен доступ к этим переменным, все равно может их изменять — случайно или злонамеренно. За исключением действительно глобальных ресурсов, таких как `cin` и `cout`, избегайте глобальных переменных. Это соответствует

принципу наименьших привилегий (principle of least privilege), который является основополагающим для грамотной разработки программного обеспечения. Суть этого принципа заключается в том, что каждому фрагменту кода можно предоставлять привилегии и доступ только в том объеме, который ему необходим для выполнения своей задачи, и не более! Примером является область видимости локальной переменной — эта переменная не должна быть видна, когда она не нужна. Локальная переменная создается при вызове функции, используется этой функцией во время ее выполнения, а затем уничтожается, когда функция завершается. Принцип наименьших привилегий делает ваши программы более надежными, предотвращая случайное или злонамеренное воздействие на переменные, которое никогда не должно происходить. Это также упрощает чтение и отладку вашего кода.

В общем случае каждую переменную надо объявлять в минимально необходимой для нее области видимости. Переменные, используемые только в какой-либо конкретной функции, должны быть объявлены как локальные переменные этой функции, а не глобальные.

Демонстрация областей видимости

На илл. 5.6 демонстрируются области видимости глобальных переменных, локальных переменных и статических локальных переменных. Для удобства обсуждения код разделен на фрагменты с примерами вывода. Для краткости мы поставили подпись только под первым фрагментом и в других аналогичных примерах будем делать так же. В строке 10 объявляется и инициализируется значением `1` глобальная переменная с именем `x`. Эта глобальная переменная будет скрыта в любом блоке и в любой функции, где объявлена переменная с этим же именем `x`.

```
 1 // fig05_06.cpp
 2 // Пример областей видимости
 3 #include <iostream>
 4 using namespace std;
 5
 6 void useLocal(); // Прототип функции
 7 void useStaticLocal(); // Прототип функции
 8 void useGlobal(); // Прототип функции
 9
10 int x{1}; // Глобальная переменная
11
```

Илл. 5.6 | Пример областей видимости

Функция main

В строке 13 выводится значение глобальной переменной `x`. В строке 15 локальная переменная `x` инициализируется значением `5`. В строке 17 локальная переменная выводится на экран, чтобы показать, что глобальная переменная `x` скрыта в функции `main`. Затем в строках 19–23 определяется новый блок в функции `main`, в ко-

тором другая локальная переменная x инициализируется значением 7 (строка 20). В строке 22 эта локальная переменная выводится на экран, чтобы показать, что она скрывает две переменные x: объявленную в функции main и глобальную. Когда блок завершается, его переменная x со значением 7 автоматически уничтожается. Далее в строке 25 выводится на экран локальная переменная x функции main, чтобы показать, что она больше не скрыта.

```
12 int main() {
13    cout << "global x in main is " << x << '\n';
14
15    const int x{5}; // Локальная переменная x в функции main
16
17    cout << "local x in main's outer scope is " << x << '\n';
18
19    { // Внутренний блок создает новую область видимости
20       const int x{7}; // Переменная скрывает ранее объявленные переменные x
21
22       cout << "local x in main's inner scope is " << x << '\n';
23    }
24
25    cout << "local x in main's outer scope is " << x << '\n';
26
```

```
global x in main is 1
local x in main's outer scope is 5
local x in main's inner scope is 7
local x in main's outer scope is 5
```

Чтобы продемонстрировать другие области видимости, мы определим в программе три функции: useLocal, useStaticLocal и useGlobal, — каждая из которых не принимает аргументы и ничего не возвращает. Остальная часть функции main (показана ниже) вызывает каждую функцию дважды в строках 27–32. После двукратного выполнения функций useLocal, useStaticLocal и useGlobal программа снова выводит на экран локальную переменную x из функции main, чтобы показать, что ни один из вызовов функций не изменил ее значение, поскольку все функции ссылались на переменные в других областях видимости.

```
27    useLocal(); // Функция с локальной переменной x
28    useStaticLocal(); // Функция с локальной статической переменной x
29    useGlobal(); // Функция, использующая глобальную переменную x
30    useLocal(); // Функция заново инициализирует свою локальную переменную x
31    useStaticLocal(); // Локальная статическая переменная x сохраняет значение
32    useGlobal(); // Глобальная переменная x сохраняет значение
33
34    cout << "\nlocal x in main is " << x << '\n';
35 }
36
```

```
local x is 25 on entering useLocal
local x is 26 on exiting useLocal
```

Окончание ⇨

```
local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
```

Функция useLocal

Функция `useLocal` инициализирует локальную переменную `x` значением `25` (строка 39). Когда `main` вызывает `useLocal` (строки 27 и 30), эта функция выводит переменную `x`, увеличивает ее и выводит снова, прежде чем вернуть управление в точку вызова. Каждый раз, когда вызывается функция `useLocal`, она заново создает локальную переменную `x` и заново инициализирует ее значением `25`.

```
37 // useLocal инициализирует переменную x при каждом вызове
38 void useLocal() {
39    int x{25}; // Инициализация при каждом вызове функции
40
41    cout << "\nlocal x is " << x << " on entering useLocal\n";
42    ++x;
43    cout << "local x is " << x << " on exiting useLocal\n";
44 }
45
```

Функция useStaticLocal

Функция `useStaticLocal` объявляет статическую переменную `x`, инициализируя ее значением `50`. Локальные статические переменные сохраняют свои значения, даже когда они находятся вне области видимости (то есть когда выполнение функции завершено). Когда в строке 28 вызывается функция `useStaticLocal`, она выводит свою локальную статическую переменную `x`, увеличивает ее и выводит снова перед возвращением управления в точку вызова. При следующем вызове этой функции (строка 31) статическая локальная переменная `x` содержит значение `51`. Инициализация в строке 50 происходит только при первом вызове функции `useStaticLocal` (строка 28).

```
46 // Функция useStaticLocal инициализирует статическую локальную переменную x
47 // только при своем первом вызове; значение переменной сохраняется
```

```
48 // между вызовами функции
49 void useStaticLocal() {
50    static int x{50}; // инициализация только при первом вызове функции
51
52    cout << "\nlocal static x is " << x
53       << " on entering useStaticLocal\n";
54    ++x;
55    cout << "local static x is " << x
56       << " on exiting useStaticLocal\n";
57 }
58
```

Функция useGlobal

Функция `useGlobal` не объявляет никаких переменных. И когда она ссылается на переменную `x`, это глобальная переменная (инициализированная в строке 10, перед функцией `main`). Когда вызывается функция `useGlobal` (в строке 29), она выводит глобальную переменную `x`, умножает ее на `10` и выводит снова, прежде чем вернуть управление в точку вызова. В следующий раз, когда `main` будут вызывать функцию `useGlobal` (в строке 32), глобальная переменная `x` будет иметь новое значение: `10`.

```
59 // useGlobal изменяет глобальную переменную x при каждом вызове
60 void useGlobal() {
61    cout << "\nglobal x is " << x << " on entering useGlobal\n";
62    x *= 10;
63    cout << "global x is " << x << " on exiting useGlobal\n";
64 }
```

5.11. Встроенные функции

Реализация программы в виде набора функций удобна с точки зрения разработчика программного обеспечения, но вызовы функций выполняются относительно медленно и создают заметную нагрузку на компьютер. Поэтому в C++, кроме обычных функций, есть **встроенные функции**, или **inline-функции (inline functions)**, которые выполняются быстрее. Спецификатор `inline` перед типом возвращаемого значения функции в ее определении указывает компилятору, что надо сделать копию основного кода функции в каждой точке ее вызова (если это необходимо), чтобы избежать многочисленных вызовов. Как правило, это увеличивает размер программы. Впрочем, компилятор может игнорировать спецификатор `inline`. Повторно используемые встроенные функции обычно помещаются в заголовки, чтобы компилятор мог встраивать их определения в каждый файл исходного кода, где они нужны.

Если вы вносите изменение в определение встроенной функции, вы должны перекомпилировать весь исходный код, в котором вызывается эта функция. Хотя компиляторы иногда могут самостоятельно оптимизировать код по принципу встроенных функций (даже если вы не используете спецификатор `inline`), в C++

Core Guidelines рекомендуется объявлять «маленькие и критичные по быстродействию» функции со спецификатором `inline`[1].

Здесь вы найдете много часто задаваемых вопросов и ответов о встроенных функциях:

https://isocpp.org/wiki/faq/inline-functions

На илл. 5.7 показана встроенная функция `cube` (строки 9–11), вычисляющая объем куба.

```
1 // fig05_07.cpp
2 // inline-функция, вычисляющая объем куба
3 #include <iostream>
4 using namespace std;
5
6 // Определение функции cube предшествует
7 // ее вызову, поэтому прототип не нужен,
8 // строка определения функции выступает в качестве прототипа
9 inline double cube(double side) {
10    return side * side * side; // Вычисляем объем куба
11 }
12
13 int main() {
14    double sideValue; // Сторона куба
15    cout << "Enter the side length of your cube: ";
16    cin >> sideValue; // Считываем число, введенное пользователем
17
18    // Вычисляем объем куба и выводим результат
19    cout << "Volume of cube with side "
20       << sideValue << " is " << cube(sideValue) << '\n';
21 }
```

```
Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875
```

Илл. 5.7 | Встроенная функция, вычисляющая объем куба

5.12. Ссылки и ссылочные параметры

Во многих языках программирования есть два способа передачи аргументов функциям: **передача по значению (pass-by-value)** и **передача по ссылке (pass-by-reference)**. При передаче по значению сначала создается копия значения аргумента, затем эта копия передается вызываемой функции. Изменения в копии не влияют на значение исходной переменной в вызывающем коде. Это предотвращает случайные побочные эффекты, которые сильно препятствуют разработке корректных и надежных программных систем. Во всех предыдущих

[1] C++ Core Guidelines. Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-inline.

примерах кода из этой книги мы передавали аргументы по значению. Но у этого способа есть недостаток: если по значению передаются большие объемы данных (большие объекты), на их копирование тратится много времени и памяти.

Ссылочные параметры функций

В этом разделе мы представим вам один из двух механизмов передачи по ссылке: ссылочные параметры[1, 2]. Когда переменная передается по ссылке, вызывающий код предоставляет вызываемой функции прямой доступ к этой переменной, поэтому функция может изменить ее в вызывающем коде.

Передача по ссылке повышает быстродействие, поскольку программа не тратит время на копирование больших объемов данных. Но передача по ссылке нарушает безопасность данных, если вызываемая функция может повредить данные в вызывающем коде..

Далее в этом разделе (после примера программы на илл. 5.8) мы покажем вам, как передавать аргумент по ссылке, чтобы получить преимущество в быстродействии (по сравнению с передачей по значению) и одновременно защитить от повреждения данные вызывающего объекта.

Ссылочный параметр (reference parameter), или **параметр-ссылка**, — это псевдоним для аргумента в вызове функции. Чтобы указать, что параметр функции передается по ссылке, добавьте амперсанд (`&`) после типа параметра и в прототипе, и в заголовке функции. Как всегда, прототип функции должен соответствовать ее заголовку. Например, следующее объявление параметра означает, что `number` — это ссылка на значение типа `int`:

```
int& number
```

В вызове такой функции, чтобы передать ей переменную по ссылке, надо просто указать имя этой переменной. В теле вызванной функции ссылочный параметр (например, `number`) ссылается на исходную переменную в вызывающем коде, которая может быть изменена этой функцией.

Передача аргументов по значению и по ссылке

На илл. 5.8 показана программа, в которой сравниваются два способа передачи аргумента: по значению (функция `squareByValue`) и по ссылке, со ссылочным параметром (функция `squareByReference`). Обратите внимание, что вызовы обеих функций синтаксически одинаковы: переменные `x` и `z` просто указаны по имени. Компилятор проверяет прототипы и определения функций, чтобы решить, когда нужна передача по значению, а когда — по ссылке.

---

[1] В главе 7 мы обсудим указатели, которые обеспечивают альтернативную форму передачи по ссылке.

[2] В главе 11 мы также обсудим другую форму ссылки, которая называется ссылкой rvalue (rvalue reference).

```
// fig05_08.cpp
// Передача аргументов по значению и по ссылке
#include <iostream>
using namespace std;

int squareByValue(int number); // Прототип с передачей по значению
void squareByReference(int& numberRef); // Прототип с передачей по ссылке

int main() {
   int x{2}; // Значение для функции squareByValue
   int z{4}; // Значение для функции squareByReference

   // Вызов функции squareByValue
   cout << "x = " << x << " before squareByValue\n";
   cout << "Value returned by squareByValue: "
      << squareByValue(x) << '\n';
   cout << "x = " << x << " after squareByValue\n\n";

   // Вызов функции squareByReference
   cout << "z = " << z << " before squareByReference\n";
   squareByReference(z);
   cout << "z = " << z << " after squareByReference\n";
}

// squareByValue возводит number в квадрат,
// сохраняет результат в number и возвращает новое значение number
int squareByValue(int number) {
   return number *= number; // Исходный аргумент не изменяется
}

// squareByReference возводит numberRef в квадрат и сохраняет результат
// в переменную, на которую ссылается numberRef, в функции main
void squareByReference(int& numberRef) {
   numberRef *= numberRef; // Исходный аргумент изменяется
}
```

```
x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

Илл. 5.8 | Передача аргументов по значению и по ссылке

Ссылки в качестве псевдонимов внутри функции

Ссылки также могут использоваться в качестве псевдонимов для других переменных внутри функции (хотя обычно они используются с функциями, как показано на илл. 5.8). Например, в следующем коде переменная `count` увеличивается через ее псевдоним `cRef`:

```
int count{1}; // Объявляем целочисленную переменную count
int& cRef{count}; // Создаем переменную cRef в качестве псевдонима для count
++cRef; // Увеличиваем переменную count (через псевдоним cRef)
```

Ссылочные переменные должны быть инициализированы в объявлениях и не могут быть переназначены в качестве псевдонимов для других переменных. В этом смысле ссылки являются константами. Все операции, выполняемые с псевдонимом (то есть со ссылкой), фактически выполняются с исходной переменной. Псевдоним — это всего лишь другое имя исходной переменной. Если это не константная ссылка (обсуждается ниже), инициализатором ссылки должно быть значение *lvalue* — то, что может отображаться в левой части присваивания, например имя переменной. Ссылка не может быть инициализирована константой или значением *rvalue*, то есть тем, что может отображаться только в правой части присваивания, например результатом вычисления.

Константные ссылки

Чтобы не разрешать ссылочному параметру изменять соответствующий аргумент в вызывающем коде, поместите константный квалификатор `const` перед именем типа в объявлении параметра. Например, рассмотрим простую функцию `displayName`:

```
void displayName(std::string name) {
   std::cout << name << '\n';
}
```

При вызове она получает копию своего аргумента `string`. Поскольку объекты `string` могут быть очень большими, эта операция копирования замедляет работу программы. По этой причине объекты `string` (и все объекты вообще) желательно передавать функциям по ссылкам. Но в данном случае функции `displayName` не нужно вносить изменения в свой аргумент. Поэтому, следуя принципу наименьших привилегий, лучше объявить параметр в виде константной ссылки:

```
const std::string& name
```

Это означает, что параметр `name` — ссылка на объект `string`, который является константой (то есть его запрещено изменять). Передавая строку по ссылке, мы ускоряем работу программы. При этом функция `displayName` считает аргумент константой, то есть не может изменить его значение в вызывающем коде.

Возврат ссылки на локальную переменную может быть опасным

При возврате ссылки на локальную нестатическую переменную эта ссылка будет указывать на переменную, уже удаленную из памяти после завершения функции. Попытка получить доступ к такой переменной приводит к неопределенному поведению, то есть к сбою программы или повреждению данных[1]. Ссылки на неопре-

[1] C++ Core Guidelines. Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-dangle.

деленные переменные называются **висячими ссылками (dangling references)**. Это логическая ошибка, из-за которой компиляторы обычно выдают предупреждение. Опытные команды разработчиков часто придерживаются политики компиляции кода с отсутствием предупреждений. Вы можете поступать так же в своей команде, используя опцию компилятора, которая интерпретирует предупреждения как ошибки.

5.13. Аргументы по умолчанию

Иногда в вашей программе функция может вызываться из нескольких точек, но с одним и тем же значением аргумента для какого-либо параметра. В таких случаях вы можете указать для этого параметра **аргумент по умолчанию (default argument)** — значение, которое будет передаваться этому параметру автоматически. Если в вызове функции не указаны аргументы для параметров, имеющих аргументы по умолчанию, компилятор переписывает этот вызов функции, вставляя в него аргументы по умолчанию.

Функция boxVolume с аргументами по умолчанию

На илл. 5.9 показан пример аргументов по умолчанию в функции `boxVolume`, вычисляющей объем параллелепипеда. В прототипе этой функции (строка 7) для всех трех параметров заданы значения по умолчанию, равные `1`.

```
1 // fig05_09.cpp
2 // Аргументы по умолчанию
3 #include <iostream>
4 using namespace std;
5
6 // Аргументы по умолчанию заданы в прототипе функции
7 int boxVolume(int length = 1, int width = 1, int height = 1);
8
9 int main() {
10    // Вызов без аргументов, используются аргументы по умолчанию
11    cout << "The default box volume is: " << boxVolume();
12
13    // Передаем аргумент length, остальные - по умолчанию
14    cout << "\n\nThe volume of a box with length 10,\n"
15       << "width 1 and height 1 is: " << boxVolume(10);
16
17    // Передаем аргументы length и width, аргумент height - по умолчанию
18    cout << "\n\nThe volume of a box with length 10,\n"
19       << "width 5 and height 1 is: " << boxVolume(10, 5);
20
21    // Передаем все три аргумента
22    cout << "\n\nThe volume of a box with length 10,\n"
23       << "width 5 and height 2 is: " << boxVolume(10, 5, 2)
24       << '\n';
25 }
```

```
26
27 // Функция boxVolume рассчитывает объем параллелепипеда
28 int boxVolume(int length, int width, int height) {
29    return length * width * height;
30 }
```

```
The default box volume is: 1

The volume of a box with length 10,
width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50

The volume of a box with length 10,
width 5 and height 2 is: 100
```

Илл. 5.9 | Аргументы по умолчанию

В первом вызове функции `boxVolume` (строка 11) аргументы не передаются, поэтому компилятор использует все три значения по умолчанию, равные `1`. Во втором вызове (строка 15) передается только аргумент `length`, а `width` и `height` получают значения по умолчанию. В третьем вызове (строка 19) передаются аргументы `length` и `width`, а `height` получает значение по умолчанию. В последнем вызове (строка 23) передаются аргументы `length`, `width` и `height`, поэтому значения по умолчанию не используются. Любые аргументы, явно переданные функции, присваиваются параметрам функции слева направо. Поэтому когда функция `boxVolume` получает один аргумент, она присваивает его значение своему параметру `length` — крайнему слева в списке. Когда `boxVolume` получает два аргумента, она присваивает их значения своим параметрам `length` и `width`. Наконец, когда функция `boxVolume` получает все три аргумента, она присваивает значения параметрам `length`, `width` и `height` соответственно.

Еще немного об аргументах по умолчанию

Аргументы по умолчанию должны быть крайними справа (завершающими) аргументами в списке параметров функции. При вызове функции с двумя или более аргументами по умолчанию, если пропущенный аргумент не является крайним справа, все аргументы справа от него тоже должны быть опущены. Аргументы по умолчанию должны быть указаны при первом появлении имени функции — как правило, в ее прототипе. Если прототип функции отсутствует и в качестве прототипа служит определение функции, аргументы по умолчанию должны быть указаны в заголовке функции. Значениями по умолчанию могут быть любые выражения, включая константы, глобальные переменные или вызовы функций. Аргументы по умолчанию можно использовать и со встроенными функциями (`inline`-функциями). Аргументы по умолчанию иногда упрощают написание вызовов функций, но некоторые программисты считают, что лучше явно указывать все аргументы, чтобы код был более понятным.

5.14. Оператор разрешения области видимости

C++ предоставляет **унарный оператор разрешения области видимости (unary scope resolution operator)** (::), необходимый для доступа к глобальной переменной, которую скрывает локальная переменная с тем же именем. Этот оператор нельзя использовать для доступа к локальной переменной с тем же именем во внешнем блоке. К глобальной переменной можно получить прямой доступ (без этого оператора), если имя глобальной переменной не совпадает с именем локальной переменной в данной области видимости.

На илл. 5.10 показано, как оператор :: позволяет различать локальную и глобальную переменные, имеющие одинаковые имена (строки 6 и 9). Чтобы подчеркнуть, что локальная и глобальная переменные `number` совершенно разные, мы объявляем одну из них с типом `int`, а другую — с типом `double`. Старайтесь не использовать переменные с одинаковыми именами для разных целей в программе. Это допускается при некоторых обстоятельствах, но часто приводит к ошибкам. Как правило, глобальных переменных надо избегать. Но если у вас в программе есть хотя бы одна глобальная переменная, всегда используйте для доступа к ней унарный оператор разрешения области видимости (::), даже если нет противоречия с именем локальной переменной. Этот оператор наглядно показывает, что вы обращаетесь к глобальной переменной, упрощает модификацию программ, снижает риск коллизий имен с локальными переменными и риск логических ошибок, которые могут возникнуть, если локальная переменная скрывает глобальную.

```
1 // fig05_10.cpp
2 // Унарный оператор разрешения области видимости
3 #include <iostream>
4 using namespace std;
5
6 const int number{7}; // Глобальная переменная number
7
8 int main() {
9    const double number{10.5}; // Локальная переменная number
10
11    // Выводим значения глобальной и локальной переменных
12    cout << "Local double value of number = " << number
13       << "\nGlobal int value of number = " << ::number << '\n';
14 }
```

```
Local double value of number = 10.5
Global int value of number = 7
```

Илл. 5.10 | Унарный оператор разрешения области видимости

5.15. Перегрузка функций

C++ позволяет определять функции с одинаковыми именами, если у них разные сигнатуры. Это называется **перегрузкой функций (function overloading)**.

Компилятор C++ выбирает подходящую функцию для вызова, проверяя количество, типы и порядок аргументов в вызове. Перегрузка функций нужна для создания нескольких одноименных функций, выполняющих аналогичные задачи с данными разных типов. Например, многие функции в математической библиотеке перегружены для разных числовых типов; согласно стандарту C++, все функции математической библиотеки, представленные в разделе 5.3, перегружены с типами `float`, `double` и `long double`. Перегрузка функций, выполняющих тесно связанные задачи, может сделать код более понятным.

Перегруженные функции square

На илл. 5.11 показаны перегруженные функции `square`, вычисляющие квадрат числа `int` (строки 7–10) и квадрат числа `double` (строки 13–16) соответственно. В строке 19 вызывается версия функции с типом `int` и ей передается литеральное значение `7`. C++ обрабатывает литеральные значения целых чисел как `int`. В строке 21 вызывается версия функции с типом `double` и ей передается литеральное значение `7.5`, которое C++ обрабатывает как `double`. В обоих случаях компилятор выбирает подходящую функцию для вызова, основываясь на типе аргумента. Выходные данные подтверждают, что в каждом случае была вызвана соответствующая функция.

```
1 // fig05_11.cpp
2 // Перегруженные функции square
3 #include <iostream>
4 using namespace std;
5
6 // Функция square для чисел int
7 int square(int x) {
8    cout << "square of integer " << x << " is ";
9    return x * x;
10 }
11
12 // Функция square для чисел double
13 double square(double y) {
14    cout << "square of double " << y << " is ";
15    return y * y;
16 }
17
18 int main() {
19    cout << square(7); // Вызываем версию для чисел int
20    cout << '\n';
21    cout << square(7.5); // Вызываем версию для чисел double
22    cout << '\n';
23 }
```

```
square of integer 7 is 49
square of double 7.5 is 56.25
```

Илл. 5.11 | Перегруженные функции square

Как компилятор различает перегруженные функции

Перегруженные функции различаются по их сигнатурам. Напомним, что сигнатура — это комбинация имени функции и ее параметров (расположенных по порядку). **Типобезопасная связь (type-safe linkage)** гарантирует вызов нужной функции с соответствием между типами аргументов и параметров. Чтобы обеспечить типобезопасную связь, компилятор внутренне кодирует идентификатор каждой функции с типами ее параметров — процесс, называемый **декорированием имен (name mangling)**. Эти кодировки варьируются в зависимости от компилятора, поэтому все, что будет связано с созданием исполняемого файла для конкретной платформы, надо компилировать для нее одним и тем же компилятором. Код на илл. 5.12 мы скомпилировали с помощью GNU C++[1]. Вместо того чтобы показывать результат выполнения программы, как мы обычно делаем, мы покажем декорированные имена функций, полученные на языке ассемблера GNU C++[2].

```
// fig05_12.cpp
// Декорирование имен для типобезопасной связи

// Функция square для чисел int
int square(int x) {
   return x * x;
}

// Функция square для чисел double
double square(double y) {
   return y * y;
}

// Функция, принимающая аргументы
// int, float, char и int&
void nothing1(int a, float b, char c, int& d) { }

// Функция, принимающая аргументы
// char, int, float& и double&
int nothing2(char a, int b, float& c, double& d) {
   return 0;
}

int main() { }
```

```
_Z6squarei
_Z6squared
_Z8nothing1ifcRi
_Z8nothing2ciRfRd
main
```

Илл. 5.12 | Декорирование имен для типобезопасной связи

[1] Отсутствие инструкций в теле функции main гарантирует, что мы не получим ошибку компоновщика, если скомпилируем этот код.

[2] Команда g++ -S fig05_12.cpp создает файл на языке ассемблера fig05_12.s.

В GNU C++ каждое декорированное имя (кроме `main`) начинается с символа подчеркивания (_), за которым следуют буква `Z`, число и имя функции. Число, следующее за `Z`, указывает количество символов в имени функции. Например, имя функции `square` содержит 6 символов, поэтому ее декорированное имя имеет префикс `_Z6`. За именем функции следует условное обозначение списка ее параметров:

- Для функции `square`, которая получает значение `int` (строка 5), буква `i` означает тип `int`, как показано в первой строке вывода.
- Для функции `square`, которая получает значение `double` (строка 10), буква `d` означает тип `double`, как показано во второй строке вывода.
- Для функции `nothing1` (строка 16) буква `i` означает тип `int`, `f` — `float`, `c` — `char`, `Ri` — `int&` (то есть ссылку на `int`), как показано в третьей строке вывода.
- Для функции `nothing2` (строка 20) буква `c` означает тип `char`, `i` — `int`, `Rf` — `float&`, `Rd` — `double&`.

Компилятор различает две функции `square` по их спискам параметров: в одном из них есть `i` для работы с числами `int`, а в другом — `d` для работы с числами `double`. Типы возвращаемых значений в декорированных именах не указаны. Перегруженные функции могут иметь разные типы возвращаемых значений, но в этом случае у них должны быть разные списки параметров. Декорирование имен функций зависит от компилятора. Например, Visual C++ создает имя `square@@YAHH@Z` для функции `square` в строке 5. Компилятор GNU C++ не декорировал имя `main`, но некоторые компиляторы это делают. Например, в Visual C++ декорированным именем будет `_main`.

Создание перегруженных функций с одинаковыми списками параметров и разными типами возвращаемых значений вызывает ошибку компиляции. Компилятор различает перегруженные функции только по спискам параметров! Такие функции не обязательно должны иметь одинаковое количество параметров.

Иногда вызов функции, имеющей аргументы по умолчанию, может быть неотличим от вызова другой перегруженной функции, и это тоже вызовет ошибку компиляции. Например, если в программе есть функция, явно не принимающая аргументы, и одноименная функция, содержащая все аргументы по умолчанию, то ошибка компиляции возникнет при попытке использовать это имя функции в вызове, не передающем аргументов. Компилятор не сможет однозначно определить, какую версию функции выбрать.

Дополнительные сведения о перегрузке функций

Все подробности о том, как компиляторы работают с вызовами перегруженных функций, вы найдете в разделе «Overload Resolution» стандарта C++:

https://timsong-cpp.github.io/cppwp/n4861/over.match

а также по этой ссылке:

https://en.cppreference.com/w/cpp/language/overload_resolution

Перегрузка операторов

В главе 11 мы расскажем о перегруженных операторах и о том, как они должны работать с объектами пользовательских типов данных. Кстати, мы уже использовали несколько перегруженных операторов, в том числе оператор вывода в поток (<<) и оператор извлечения из потока (>>), которые перегружены для всех основных типов данных. В главе 11 мы подробнее обсудим перегрузку операторов << и >> для обработки объектов пользовательских типов.

5.16. Шаблоны функций

Как правило, перегруженные функции выполняют аналогичные операции с разными типами данных. Если логика программы и выполняемые операции идентичны для каждого типа данных, перегрузку можно выполнить компактнее и удобнее с помощью **шаблона функции (function template)**. Вы пишете только одно определение шаблона функции. Затем компилятор C++, учитывая разные типы аргументов, предоставляемых в вызовах этой функции, автоматически генерирует отдельные **специализации шаблона (template specializations)** этой функции, также называемые **экземплярами шаблона (template instantiations)**, для соответствующей обработки каждого вызова. Таким образом, определение одного шаблона функции фактически определяет целое семейство перегруженных функций. Программирование с шаблонами называют **обобщенным программированием (generic programming)**. В этом разделе мы определим пользовательский шаблон функции. В последующих главах вам встретятся многие шаблоны функций стандартной библиотеки C++. В главе 15 мы подробно рассмотрим пользовательские шаблоны, включая сокращенные шаблоны функций и концепты, введенные в C++20.

Шаблон функции maximum

На илл. 5.13 показан шаблон функции `maximum`, которая находит наибольшее из трех значений. Определение шаблона функции всегда начинается с ключевого слова `template` (строка 3), за которым следует список параметров шаблона в угловых скобках (< и >). В любом из этих параметров (в одном или в нескольких) основной или пользовательский тип данных заменяется заполнителем — **параметром-типом (type parameter)**, которому предшествует ключевое слово `typename` или `class` (в данном контексте это синонимы). В нашем примере список параметров состоит из одного параметра-типа с именем `T`. Такими заполнителями обозначаются: типы параметров функции (строка 4), тип возвращаемого значения функции (строка 4) и типы переменных в теле определения функции (строка 5). Шаблон функции определяется так же, как и любая другая функция, но вместо одного или нескольких реальных типов данных в шаблоне подставляются параметры-типы.

Параметр-тип `T` в шаблоне функции `maximum` (строка 3) является заполнителем для типа данных, обрабатываемых этой функцией. Встречая вызов функции `maximum`

в исходном коде программы, компилятор подставляет соответствующий тип аргументов вместо каждого заполнителя T в определении шаблона. Так создается полная специализация шаблона нашей функции, определяющая наибольшее из трех значений указанного типа. Все три значения должны быть одного типа, поскольку в этом примере мы используем только один параметр-тип. Затем вновь созданная функция компилируется — шаблоны являются средством генерации кода. В главе 13 мы будем использовать шаблоны стандартной библиотеки C++, имеющие несколько параметров-типов.

```
1 // Fig. 5.13: maximum.h
2 // Заголовочный файл с шаблоном функции maximum
3 template <typename T> // Или template <class T>
4 T maximum(T value1, T value2, T value3) {
5    T maximumValue{value1}; // Предположим, что число value1 - наибольшее
6
7    // Проверяем: число value2 больше, чем maximumValue?
8    if (value2 > maximumValue) {
9       maximumValue = value2;
10   }
11
12   // Проверяем: число value3 больше, чем maximumValue?
13   if (value3 > maximumValue) {
14      maximumValue = value3;
15   }
16
17   return maximumValue;
18 }
```

Илл. 5.13 | Заголовочный файл с шаблоном функции maximum

Тестирование шаблона функции maximum

На илл. 5.14 показана программа, которая с помощью шаблона функции **maximum** находит наибольшее из трех значений **int**, трех значений **double** и трех значений **char** соответственно (строки 15, 24 и 33). Поскольку типы аргументов в каждом из трех вызовов отличаются, компилятор «под капотом» создает для этих вызовов отдельные определения функции: первое ожидает три значения **int**, второе — три значения **double**, третье — три значения **char**.

```
1 // fig05_14.cpp
2 // Тестирование шаблона функции maximum
3 #include <iostream>
4 #include "maximum.h" // Определение шаблона функции maximum
5 using namespace std;
6
7 int main() {
8    // Вводим три значения int
9    cout << "Input three integer values: ";
```

Окончание ⇨

```
10    int int1, int2, int3;
11    cin >> int1 >> int2 >> int3;
12
13    // Вызываем функцию maximum с типом int
14    cout << "The maximum integer value is: "
15       << maximum(int1, int2, int3);
16
17    // Вводим три значения double
18    cout << "\n\nInput three double values: ";
19    double double1, double2, double3;
20    cin >> double1 >> double2 >> double3;
21
22    // Вызываем функцию maximum с типом double
23    cout << "The maximum double value is: "
24       << maximum(double1, double2, double3);
25
26    // Вводим три значения char
27    cout << "\n\nInput three characters: ";
28    char char1, char2, char3;
29    cin >> char1 >> char2 >> char3;
30
31    // Вызываем функцию maximum с типом char
32    cout << "The maximum character value is: "
33       << maximum(char1, char2, char3) << '\n';
34 }
```

```
Input three integer values: 1 2 3
The maximum integer value is: 3

Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3

Input three characters: A C B
The maximum character value is: C
```

Илл. 5.14 | Тестирование шаблона функции maximum

Специализация шаблона функции maximum для типа int

В специализации шаблона функции, созданной для работы с типом `int`, компилятор заменяет все заполнители `T` на `int`:

```
int maximum(int value1, int value2, int value3) {
   int maximumValue{value1}; // Предположим, что число value1 - наибольшее

// Проверяем: число value2 больше, чем maximumValue?
if (value2 > maximumValue) {
   maximumValue = value2;
}
// Проверяем: число value3 больше, чем maximumValue?
```

```
   if (value3 > maximumValue) {
      maximumValue = value3;
   }
   return maximumValue;
}
```

5.17. Рекурсия

Для решения некоторых видов задач хорошо подходят функции, вызывающие сами себя. Рекурсивная функция — это функция, которая вызывает саму себя прямо или косвенно (через другую функцию). В этом и в следующем разделах представлены простые примеры рекурсии. Более детальное изучение рекурсии входит в курс информатики высшего уровня.

Концепция рекурсии

Сначала мы рассмотрим рекурсию концептуально, затем перейдем к программам с рекурсивными функциями. Рекурсивные подходы к решению задач имеют несколько общих элементов. Рекурсивная функция вызывается для решения задачи, но умеет решать только ее простейший вариант: так называемую **базовую задачу (base case)**. Если функция вызывается с базовой задачей, она сразу возвращает результат. Если функция вызывается с более сложной задачей, она делит эту задачу на две концептуальные части: 1) часть, которую она может решить, и 2) часть, которую она не может решить. Для правильной рекурсии вторая часть должна быть упрощенной (уменьшенной) версией исходной задачи. Упрощенная задача похожа на исходную, поэтому функция вызывает свою копию для работы над ней. Это называется **рекурсивным вызовом (recursive call)**, или **шагом рекурсии (recursion step)**. Шаг рекурсии часто включает ключевое слово `return`, потому что результат шага будет объединен с той частью задачи, которую функция может решить, — для формирования результата, возвращаемого в точку вызова (например, в функцию `main`).

Недостижимость базовой задачи или неправильно написанный шаг рекурсии (без сходимости к базовой задаче) приводят к ошибке в виде бесконечной рекурсии, которая обычно завершается крахом программы из-за переполнения стека. Бесконечная рекурсия также возникает, если функция, которая не должна была быть рекурсивной, случайно вызывает саму себя, прямо или косвенно (через другую функцию).

Шаг рекурсии выполняется, пока «открыт» исходный вызов функции, то есть пока его выполнение еще не завершилось. Шаг рекурсии может привести к множеству последующих шагов, поскольку функция продолжает разделять каждую новую подзадачу, с которой она вызывается, на две концептуальные части. Чтобы рекурсия в итоге завершилась, функция при каждом вызове самой себя должна получать все более простую версию задачи. Иными словами, последовательность упрощающихся задач должна постепенно сходиться к базовой задаче. Когда функ-

ция распознаёт базовую задачу, она возвращает результат в предыдущую копию функции. Затем происходит последовательность возвратов — до тех пор, пока исходный вызов функции не вернет конечный результат в точку вызова[1]. Все это выглядит довольно экзотично по сравнению с теми способами решения задач, которые вы уже знаете. В качестве примера рекурсии мы рассмотрим программу для популярных математических вычислений.

Факториал

Напомним, что факториал — это произведение всех целых чисел от 1 до заданного числа. Факториал неотрицательного целого числа n, обозначаемый в математической записи как $n!$, является результатом выражения:

$$n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

При этом принимается, что факториал нуля равен 1 ($0! = 1$), факториал единицы тоже равен 1 ($1! = 1$). Например, 5! (произносится «пять факториал» или «факториал числа пять») — это результат выражения $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, что составляет 120.

Итеративное вычисление факториала

Факториал целого числа `number`, большего или равного 0, может быть вычислен итеративно (не рекурсивно) в цикле `for`:

```
int factorial{1};
for (int counter{number}; counter >= 1; --counter) {
   factorial *= counter;
}
```

Рекурсивное вычисление факториала

Рекурсивное определение факториала достигается разбором следующего алгебраического выражения:

$$n! = n \cdot (n-1)!$$

Например, очевидно, что $5! = 5 \cdot 4!$, как показано в следующем примере:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$

$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$

$$5! = 5 \cdot (4!)$$

Вычисление 5!

На следующей диаграмме вы видите процесс вычисления выражения 5! рекурсивным методом. Часть (а) показывает, как продолжается последовательность

[1] В стандарте C++ указано, что функцию main не следует вызывать из программы или рекурсивно. Единственное назначение этой функции — быть стартовой точкой для выполнения программы. «Стандарт C++». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/basic.start.main и https://timsong-cpp.github.io/cppwp/n4861/expr.call.

рекурсивных вызовов, пока не дойдет до вычисления факториала единицы (1! = 1), что завершит рекурсию. Часть (b) показывает значения, которые будут возвращаться в каждом рекурсивном вызове, пока не будет вычислено и возвращено конечное значение.



Вычисление факториалов с помощью рекурсивной функции factorial

На илл. 5.15 показана программа, которая вычисляет и выводит факториалы чисел от `0` до `10`. Рекурсивная функция `factorial` (строки 18–25) сначала проверяет условие `number <= 1`, означающее базовую задачу (строка 19). Если `number` меньше или равно `1`, функция возвращает `1` (строка 20), дальнейшая рекурсия не требуется и функция завершается. Если `number` больше `1`, в строке 23 задача выражается в виде произведения `number` и результата рекурсивного вызова функции `factorial`, вычисляющего факториал выражения `number - 1`. Это вычисление является немного более простой задачей, чем исходное вычисление `factorial(number)`.

```
1 // fig05_15.cpp
2 // Рекурсивная функция factorial
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 long factorial(int number); // Прототип функции
8
```

Окончание ⇨

```
 9 int main() {
10    // Вычисляем факториалы чисел от 0 до 10
11    for (int counter{0}; counter <= 10; ++counter) {
12    cout << setw(2) << counter << "! = " << factorial(counter)
13       << '\n';
14    }
15 }
16
17 // Рекурсивное определение функции factorial
18 long factorial(int number) {
19    if (number <= 1) { // Проверяем базовую задачу
20       return 1; // 0! = 1 и 1! = 1
21    }
22    else { // Делаем шаг рекурсии
23       return number * factorial(number - 1);
24    }
25 }
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

Илл. 5.15 | Рекурсивная функция, вычисляющая факториалы

Значения факториалов быстро увеличиваются

Функция `factorial` принимает параметр типа `int` и возвращает результат типа `long`. Обычно `long` хранится по меньшей мере в четырех байтах (32 бита); такая переменная может содержать значение в диапазоне от `-2 147 483 648` до `2 147 483 647`. К сожалению, функция `factorial` возвращает числа, возрастающие так быстро, что тип `long` не позволяет вычислить много факториалов до того, как мы выйдем за пределы диапазона `long`. Для вычислений с еще большими числами мы могли бы использовать тип `long long` (раздел 3.11) или класс, представляющий целые числа произвольного размера (например, класс `BigNumber` с открытым исходным кодом, который мы представили в разделе 3.12).

5.18. Пример рекурсивных вычислений: числа Фибоначчи

Ряд чисел Фибоначчи начинается так:

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Иными словами, первые числа ряда — это `0` и `1`, а каждое последующее число Фибоначчи является суммой двух предыдущих.

Эти числа можно найти в природе, так как они описывают форму спирали. Отношение между двумя любыми соседними элементами ряда Фибоначчи сходится к числу `1.618...`, представленному константой `numbers::phi` в заголовке `<numbers>`. Это число называется «золотым сечением» или «золотой пропорцией». Люди считают эту пропорцию эстетически приятной. Архитекторы часто проектируют окна, комнаты и здания, стороны которых имеют именно такую пропорцию. Она встречается даже в соотношении длины и ширины почтовых открыток. По запросу «числа Фибоначчи в природе» вы найдете в интернете много интересных примеров, включая лепестки цветов, раковины, ураганы и спиральные галактики.

Определение рекурсивной функции fibonacci

Последовательность чисел Фибоначчи может быть определена рекурсивно таким образом:

$$\text{fibonacci}(0) = 0$$

$$\text{fibonacci}(1) = 1$$

$$\text{fibonacci}(n) = \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)$$

В программе, показанной на илл. 5.16, для рекурсивного вычисления числа Фибоначчи с порядковым номером n используется функция `fibonacci`. Программа выводит числа Фибоначчи с несколькими порядковыми номерами. Как видите, числа в ряду Фибоначчи быстро возрастают, хотя и гораздо медленнее, чем факториалы.

```
// fig05_16.cpp
// Рекурсивная функция fibonacci
#include <iostream>
using namespace std;

long fibonacci(long number); // Прототип функции

int main() {
   // Вычисляем числа Фибоначчи от 1 до 10
   for (int counter{0}; counter <= 10; ++counter)
      cout << "fibonacci(" << counter << ") = "
         << fibonacci(counter) << '\n';

   // Выводим 20-е, 30-е и 35-е числа Фибоначчи
   cout << "\nfibonacci(20) = " << fibonacci(20) << '\n';
   cout << "fibonacci(30) = " << fibonacci(30) << '\n';
   cout << "fibonacci(35) = " << fibonacci(35) << '\n';
}

```

Окончание ⇨

```
20 // Рекурсивная функция fibonacci
21 long fibonacci(long number) {
22    if ((0 == number) || (1 == number)) { // Проверяем базовую задачу
23       return number;
24    }
25    else { // Делаем шаг рекурсии
26       return fibonacci(number - 1) + fibonacci(number - 2);
27    }
28 }
```

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(2) = 1
fibonacci(3) = 2
fibonacci(4) = 3
fibonacci(5) = 5
fibonacci(6) = 8
fibonacci(7) = 13
fibonacci(8) = 21
fibonacci(9) = 34
fibonacci(10) = 55

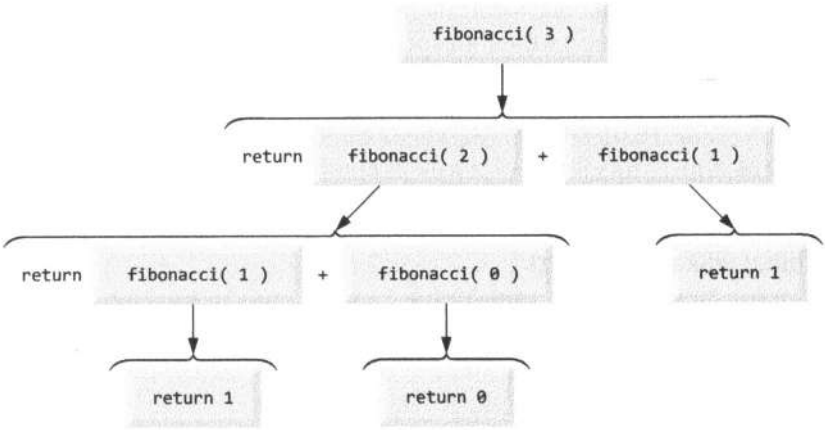
fibonacci(20) = 6765
fibonacci(30) = 832040
fibonacci(35) = 9227465
```

Илл. 5.16 | Рекурсивная функция fibonacci

Программа начинается с цикла, который вычисляет и отображает числа Фибоначчи в диапазоне от `0` до `10`. Далее следуют три вызова функции для вычисления чисел Фибоначчи с номерами `20`, `30` и `35` (строки 15–17). Вызовы `fibonacci` в функции `main` (строки 12 и 15–17) не являются рекурсивными, но вызовы из функции `fibonacci` (строка 26) являются рекурсивными. Каждый раз, когда программа вызывает функцию `fibonacci` (строки 21–28), эта функция проверяет базовую задачу, сравнивая `number` с нулем и единицей (строка 22). Если условие истинно, в строке 23 возвращается `number`. Интересный момент: если `number` больше `1`, шаг рекурсии (строка 26) генерирует два рекурсивных вызова, каждый для немного более простой задачи, чем исходный вызов функции `fibonacci`.

Вычисление fibonacci(3)

На следующей диаграмме показано, как функция `fibonacci` вычисляет число Фибоначчи с номером `3`. Возникает интересный вопрос о том порядке, в котором компиляторы C++ вычисляют операнды операторов. В данном случае правила приоритета и группировки операторов не дают ответа; это отдельная проблема. Как видите, вычисление `fibonacci(3)` влечет за собой два рекурсивных вызова функции: `fibonacci(2)` и `fibonacci(1)`. В каком порядке будут выполнены эти вызовы?



Порядок вычисления операндов

Большинство программистов предполагают, что операнды вычисляются слева направо, что имеет место в некоторых языках программирования. На самом деле в C++ не регламентируется порядок вычисления операндов многих операторов (в том числе `+`). Поэтому вы не должны делать никаких предположений о порядке выполнения этих вызовов. Программа может сначала выполнить `fibonacci(2)`, а потом `fibonacci(1)` или наоборот: сначала `fibonacci(1)`, а потом `fibonacci(2)`. В нашей программе, как и в большинстве других, конечный результат в любом случае будет одним и тем же. Однако в некоторых программах вычисление операндов вызывает побочные эффекты (например, операции ввода-вывода или изменения значений переменных), которые могут повлиять на конечный результат выражения.

Операторы, для которых указан порядок вычисления

До C++17 в языке C++ регламентировался порядок вычисления операндов только для четырех операторов: `&&`, `||`, «запятая» (`,`) и `?:`. Первые три — это бинарные операторы, два операнда которых гарантированно вычисляются слева направо. Четвертый является единственным оператором C++ с тремя операндами. Его крайний левый операнд всегда вычисляется первым. Если он принимает значение `true`, средний операнд вычисляется следующим, а последний операнд игнорируется. Если крайний левый операнд вычисляется как `false`, третий операнд вычисляется следующим, а средний операнд игнорируется.

Начиная с C++17, регламентируется порядок вычисления операндов еще для нескольких операторов. Компилятор вычисляет операнды слева направо для операторов `[]` (глава 6), `->` (глава 7), `<<`, `>>`, `.*`, `->*`, а также для оператора «круглые скобки» в вызове функции (это означает, что компилятор обрабатывает имя функции раньше, чем ее аргументы). Операнды операторов присваивания компилятор вычисляет справа налево.

Если программа написана так, что зависит от порядка вычисления операндов, это может привести к логическим ошибкам. Для уверенности в том, что побочные эффекты проявляются в нужном порядке, разбивайте сложные выражения на отдельные инструкции. Напомним, что операторы `&&` и `||` используют вычисления по короткой схеме. Размещение выражения с побочным эффектом в правой части оператора `&&` или `||` является логической ошибкой, если это выражение всегда должно вычисляться.

Экспоненциальная сложность

Рекурсивные программы, подобные той, которую мы написали для генерации чисел Фибоначчи, могут быть опасными. Дело в том, что каждый шаг рекурсии в функции `fibonacci` удваивает количество вызовов функции. Таким образом, количество рекурсивных вызовов, необходимых для вычисления числа Фибоначчи с порядковым номером n, составляет 2^n. Этот стремительный рост быстро выходит из-под контроля. Вычисление 20-го числа Фибоначчи требует 2^{20} (около миллиона) вызовов. Вычисление 30-го числа Фибоначчи требует уже 2^{30} (около миллиарда) вызовов, и т. д. ИТ-специалисты называют это **экспоненциальной сложностью (exponential complexity)**. Задачи такого рода с большими значениями n могут оказаться чрезмерной нагрузкой даже для самых мощных компьютеров в мире. Вопросы сложности (в том числе экспоненциальной сложности) подробно изучаются в курсе информатики высшего уровня, который обычно называется «Алгоритмы». Избегайте рекурсивных программ «в стиле Фибоначчи», которые приводят к экспоненциальному «взрыву» вызовов.

5.19. Рекурсия или итерация?

В предыдущих разделах мы рассмотрели две рекурсивные функции. Их также можно реализовать с помощью простых итеративных программ. В этом разделе мы сравним оба подхода, чтобы вам было проще выбирать лучший из них в зависимости от ситуации.

- Итерация и рекурсия основаны на управляющих инструкциях. Для итерации программист использует инструкцию цикла. Для рекурсии программист использует инструкцию выбора (ветвления).
- Итерация и рекурсия основаны на многократном повторении. При итерации повторение реализуется инструкцией цикла. При рекурсии повторение реализуется функцией, которая вызывает саму себя.
- Итерация и рекурсия завершаются при определенном условии. Итерация завершается, когда перестает выполняться условие продолжения цикла. Рекурсия завершается, когда распознается базовая задача.
- Итерация с управляющей переменной (цикл со счетчиком) и рекурсия постепенно приближаются к завершению. Итерация изменяет счетчик до тех пор, пока он не примет значение, приводящее к невыполнению условия продол-

жения цикла. Рекурсия создает более простые версии исходной задачи до тех пор, пока не будет достигнута базовая задача.

- Итерация и рекурсия могут выполняться бесконечно. Бесконечный цикл возникает, если условие продолжения никогда не становится ложным. Бесконечная рекурсия возникает, если шаг рекурсии не упрощает задачу в каждом рекурсивном вызове таким образом, чтобы в итоге достичь базовой задачи.

Итеративная версия функции factorial

Давайте рассмотрим различия между итерацией и рекурсией с помощью итеративной программы, вычисляющей факториалы (илл. 5.17). В строках 22–24 используется инструкция цикла, в отличие от инструкции выбора в нашей прежней рекурсивной программе (илл. 5.15, строки 19–24). В рекурсивной программе (илл. 5.15) в строке 19 проверяется, достигнута ли базовая задача, завершающая рекурсию. В итеративной программе (илл. 5.17) в строке 22 проверяется условие продолжения цикла, и если оно не выполняется, цикл завершается. Кроме того, итеративная программа не создает упрощенные версии исходной задачи, а вместо этого выполняет инкремент счетчика до тех пор, пока не станет ложным условие продолжения цикла.

```
 1 // fig05_17.cpp
 2 // Итеративная версия функции factorial
 3 #include <iostream>
 4 #include <iomanip>
 5 using namespace std;
 6
 7 long factorial(int number); // Прототип функции
 8
 9 int main() {
10    // Вычисляем факториалы чисел от 0 до 10
11    for (int counter{0}; counter <= 10; ++counter) {
12       cout << setw(2) << counter << "! = " << factorial(counter)
13          << '\n';
14    }
15 }
16
17 // Итеративная функция factorial
18 long factorial(int number) {
19    long result{1};
20
21    // Итеративное вычисление факториала
22    for (int i{number}; i >= 1; --i) {
23       result *= i;
24    }
25
26    return result;
27 }
```

```
0! = 1
1! = 1
```

Окончание ⇨

```
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

Илл. 5.17 | Итеративная версия функции factorial

Недостатки рекурсии

При рекурсии многократно активируется ресурсоемкий механизм вызова функций, поэтому такая программа может занять много процессорного времени и памяти. Каждый рекурсивный вызов создает очередную копию переменных функции; в итоге объем занятой памяти может оказаться чрезмерным. Итерация обычно выполняется внутри функции, поэтому памяти требуется значительно меньше и процессорное время на многократные вызовы функций тоже не расходуется. Но почему же программисты все-таки иногда выбирают рекурсию?

Когда рекурсия предпочтительнее итерации?

Любую задачу, решаемую рекурсивно, можно решить и итеративно (без рекурсии). Рекурсию обычно выбирают для упрощения кода — в тех случаях, когда рекурсивный алгоритм получается нагляднее и проще в отладке, чем итеративный. Кроме того, рекурсию выбирают, когда для задачи уже есть очевидное рекурсивное решение, а итеративного решения нет или его трудно найти. По возможности избегайте рекурсии в программах, которые должны работать быстро. Не забывайте, что рекурсивные вызовы требуют много времени и памяти.

5.20. Lnfylun Lhqtomh Wjtz Qarcv: Qjwazkrplm xzz Xndmwwqhlz

Конечно, вы заметили, что последний пункт в списке целей главы 5, последняя фраза раздела 5.1 и заголовок раздела 5.20 — все это выглядит как тарабарщина. Но это не ошибки! В этом разделе мы продолжаем презентации готовых объектов. Вы научитесь шифровать и расшифровывать сообщения с помощью объекта, который легко создадите из нашего класса, реализующего шифр Виженера с секретным ключом[1].

В предыдущих главах вы создавали объекты класса `string` стандартной библиотеки C++ и объекты других классов, входящих в библиотеки с открытым исходным кодом. Вероятно, вы когда-нибудь будете работать и с классами, написанными

[1] «Vigenère Cipher». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Vigenère_cipher.

вашей организацией или вашей командой для внутренних нужд или для конкретного проекта. А для этого раздела мы написали наш собственный класс `Cipher` (он находится в заголовке `"cipher.h"`) и предоставили его вам. В главе 9 вы будете создавать свои собственные классы.

Криптография

Криптография существует уже тысячи лет[1,2] и имеет большое значение в современном мире. Каждый день криптографические алгоритмы работают «за кулисами» интернета, обеспечивая конфиденциальность и безопасность ваших коммуникаций. Например, при обмене данными между большинством веб-сайтов и вашим браузером выполняется шифрование и дешифрование по протоколу HTTPS.

Шифр Цезаря

Юлий Цезарь использовал простой **шифр подстановки (substitution cipher)** для шифрования военных сообщений[3]. Его способ, известный как шифр Цезаря, заключается в замещении каждой буквы в сообщении на букву, находящуюся на три позиции впереди в алфавите. Буква A заменяется на D, B — на E, C — на F, ... X — на A, Y — на B и Z — на C. Например, текст:

```
Caesar Cipher
```

в зашифрованном виде будет выглядеть так:

```
Fdhvdu Flskhu
```

Зашифрованный текст называют криптограммой, или шифротекстом.

Если вам интересно поиграть с шифром Цезаря и со многими другими методами шифрования, посетите веб-сайт:

https://cryptii.com/pipes/caesar-cipher

который является онлайн-реализацией библиотеки `cryptii` с открытым исходным кодом:

https://github.com/cryptii/cryptii

Шифр Виженера

Простейшие шифры подстановки, такие как шифр Цезаря, относительно легко расшифровать. Например, буква «e» встречается в английском языке чаще всех остальных. Вы можете изучить текст, зашифрованный шифром Цезаря, и с большой вероятностью предположить, что наиболее часто встречающийся символ является буквой «e».

---

[1] «Cryptography». 14 мая 2020 г. https://en.wikipedia.org/wiki/Cryptography#History_of_cryptography_and_cryptanalysis.

[2] «History of Cryptography». 19 января 2020 г. https://www.binance.vision/security/history-of-cryptography.

[3] «Шифр Цезаря». Был доступен: 15 сентября 2023 г. https://ru.wikipedia.org/wiki/Шифр_Цезаря.

В следующем примере мы покажем вам шифр Виженера, который является шифром подстановки с секретным ключом. Этот метод аналогичен одновременному применению 26 шифров Цезаря — по одному на каждую букву алфавита. Шифр Виженера использует буквы из исходного текста и секретный ключ для поиска в соответствующих шифрах Цезаря заменяющих символов. Здесь вы можете узнать больше о реализации этого метода:

https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher

Секретный ключ этого шифра должен состоять только из букв. Как и пароли, этот ключ не должен быть легко угадываемым. Например, мы выбрали ключ из 11 случайно выбранных символов:

```
XMWUJBVYHXZ
```

Количество символов, которые вы можете использовать в своем секретном ключе, не ограниченно. Ваш адресат, который получит зашифрованный текст, должен знать тот секретный ключ, который вы использовали при шифровании[1]. Поэтому вы должны сообщить ключ заранее, например, при личной встрече.

Программа с нашим классом Cipher

В программе, показанной на илл. 5.18, вы будете использовать класс `Cipher`, который реализует шифр Виженера. Заголовок `"cipher.h"` (строка 3) из папки примеров `ch05` этой главы определяет наш класс. Вам даже не нужно читать и понимать исходный код, чтобы использовать средства шифрования и дешифрования. Вы просто создаете объект класса `Cipher`, а затем вызываете его функции `encrypt` и `decrypt` для шифрования и дешифрования текста соответственно.

```
1 // fig15_18.cpp
2 // Шифрование и дешифрование текста шифром Виженера
3 #include "cipher.h"
4 #include <iostream>
5 #include <string>
6 using namespace std;
7
8 int main() {
9    string plainText;
10   cout << "Enter the text to encrypt:\n";
11   getline(cin, plainText);
12
13   string secretKey;
14   cout << "\nEnter the secret key:\n";
15   getline(cin, secretKey);
16
17   Cipher cipher;
```

[1] Есть много веб-сайтов, предлагающих восстановить зашифрованный шифром Виженера текст без оригинального секретного ключа. Мы попробовали несколько вариантов, но ни один из них не показал наш исходный текст.

```
18
19    // Шифрование plainText с помощью secretKey
20    string cipherText{cipher.encrypt(plainText, secretKey)};
21    cout << "\nEncrypted:\n " << cipherText << '\n';
22
23    // Дешифрование cipherText
24    cout << "\nDecrypted:\n "
25       << cipher.decrypt(cipherText, secretKey) << '\n';
26
27    // Дешифрование текста, введенного пользователем
28    cout << "\nEnter the ciphertext to decipher:\n";
29    getline(cin, cipherText);
30    cout << "\nDecrypted:\n "
31       << cipher.decrypt(cipherText, secretKey) << '\n';
32 }
```

```
Enter the text to encrypt:
Welcome to Modern C++ application development with C++20!

Enter the secret key:
XMWUJBVYHXZ

Encrypted:
Tqhwxnz rv Jnaqnh L++ bknsfbxfeiw eztlinmyahc xdro Z++20!

Decrypted:
Welcome to Modern C++ application development with C++20!

Enter the ciphertext to decipher:
Lnfylun Lhqtomh Wjtz Qarcv: Qjwazkrplm xzz Xndmwwqhlz

Decrypted:
Objects Natural Case Study: Encryption and Decryption
```

Илл. 5.18 | Шифрование и дешифрование текста шифром Виженера

Функции класса Cipher

Класс `Cipher` предоставляет вам две основные функции:

- Функция `encrypt` получает строки, представляющие исходный текст и секретный ключ. Используя шифр Виженера, она возвращает строку с зашифрованным текстом.
- Функция `decrypt` получает строки, представляющие зашифрованный текст и секретный ключ. Используя шифр Виженера, она выполняет обратную операцию и возвращает строку с расшифрованным текстом.

Сначала программа просит ввести текст для шифрования и секретный ключ. В строке 17 создается объект `Cipher`. В строках 20–21 введенный текст зашифровывается и выводится на экран. В строках 24–25 полученная криптограмма расшифровывается, чтобы показать вам исходный текст.

Теперь вы можете узнать, что написано в последнем пункте в списке целей главы 5, в последней фразе раздела 5.1 и в заголовке раздела 5.20. Это криптограммы, которые мы получили с помощью нашего класса `Cipher` и секретного ключа:

```
XMWUJBVYHXZ
```

В строках 28–29 запрашивается зашифрованный текст, затем в строках 30–31 он расшифровывается и на экран выводится исходный текст.

5.21. Итоги

В этой главе мы рассказали о функциях, в том числе о прототипах, сигнатурах, заголовках и телах функций. Мы рассмотрели функции математической библиотеки и математические функции и константы, добавленные в C++11, C++17 и C++20.

Вы узнали о принудительном приведении аргументов к соответствующим типам, указанным в объявлении параметров функции. Мы представили обзор заголовков стандартной библиотеки C++, метод генерации недетерминированных случайных чисел (C++11), перечисления `enum` с ограниченной областью видимости и объявления `using enum` (C++20).

Вы узнали об областях видимости переменных. Мы обсудили два способа передачи аргументов функциям: по значению и по ссылке. Мы рассказали о встроенных функциях и о функциях с аргументами по умолчанию. Вы узнали, что перегруженные функции имеют одно и то же имя, но разные сигнатуры. Такие функции могут выполнять одинаковые или похожие задачи, получая разные типы данных или разное количество параметров. Мы продемонстрировали применение шаблонов для удобного создания семейств перегруженных функций. Мы рассказали о рекурсивных вызовах функций для решения сложных задач. Затем мы представили очередной пример работы с готовыми объектами: шифрование и дешифрование текста шифром Виженера.

В главе 6 вы узнаете о списках и таблицах данных в массивах и объектно-ориентированных векторах. Вы также увидите более элегантную реализацию программы для игры в кости, основанную на массиве.

6

Массивы, векторы, библиотека ranges и функциональное программирование

В этой главе вы:

- познакомитесь с массивами — упорядоченными наборами данных, имеющими фиксированные размеры;
- научитесь объявлять массивы, инициализировать их и обращаться к их элементам;
- узнаете, как инструкция `for` для диапазона значений помогает предотвращать ошибки итерации;
- увидите несколько примеров передачи массива функциям;
- отсортируете элементы массива в порядке возрастания;
- выполните быстрый поиск в отсортированном массиве с помощью функции `binary_search`;
- научитесь объявлять и использовать многомерные массивы;
- узнаете о возможностях функционального программирования с библиотекой `ranges` (C++20);
- продолжите работать с готовыми объектами, на этот раз с векторами (`vector`) — упорядоченными наборами данных, динамически меняющими свои размеры.

6.1. Введение

В этой главе мы представим вам **структуры данных (data structures)**, то есть упорядоченные наборы данных, состоящие из однотипных элементов. Структуры данных, реализованные современными средствами стандартной библиотеки C++, называются **контейнерами (containers)**. Мы рассмотрим два вида контейнеров:

- **массив (array)**[1] — контейнер фиксированного размера, основанный на шаблоне класса `array` (он появился в C++11);
- **вектор (vector)** — контейнер переменного размера, основанный на шаблоне класса `vector`.

В отличие от массива, размер вектора (количество его элементов) может увеличиваться и уменьшаться во время выполнения программы. Чтобы использовать в программе эти контейнеры, надо включить в нее заголовки `<array>` и `<vector>` соответственно.

Прежде всего мы расскажем, как объявлять, создавать и инициализировать массивы, и приведем примеры операций с ними. Затем мы займемся сортировкой элементов массива в порядке возрастания и поиском данных в отсортированном массиве. Вы увидите, что попытка доступа к элементу данных за пределами массива или вектора (то есть к несуществующему элементу) вызывает исключение — ошибку во время выполнения программы. Чтобы программа корректно реагировала на ошибку, мы обработаем это исключение. В главе 12 исключения рассматриваются подробнее.

На языке C++ (как и на многих других современных языках) можно программировать «в функциональном стиле». Это помогает писать более лаконичный код, снижать вероятность ошибок, делать код нагляднее и удобнее для отладки и модификации. Такой код проще распараллеливать для многоядерных процессоров. Мы покажем примеры функционального программирования с библиотекой `ranges`, введенной в C++20. Наконец, мы продолжим презентацию готовых объектов, которые на этот раз будем создавать на основе шаблона класса `vector` стандартной библиотеки C++. Таким образом, после изучения этой главы вы будете уметь работать с двумя важнейшими структурами данных, имеющими много общего: с массивами (`array`) и векторами (`vector`).

Полностью квалифицированные имена стандартной библиотеки

Начиная с раздела 2.6, мы включали директиву `using namespace std;` в каждую программу, поэтому нам не нужно было добавлять квалификатор `std::` к каждому имени стандартной библиотеки C++. Но в больших программах директива `using` может привести к скрытым ошибкам, которые трудно обнаружить. Поэтому

[1] В главе 7 вы увидите, что C++ также поддерживает традиционные массивы в стиле языка C, которые не являются контейнерами. Но в современных программах на C++ вместо них необходимо использовать массивы, основанные на шаблоне класса array из C++11.

в дальнейшем мы будем **полностью квалифицировать (fully qualify)** большинство идентификаторов стандартной библиотеки C++, то есть записывать имена с квалификатором `std::`.

6.2. Массивы

Элементы массива (хранящиеся в нем элементы данных) расположены в памяти последовательно. На следующей диаграмме показан целочисленный массив с именем `c`, содержащий пять элементов:



Для обращения к элементу массива надо указать имя массива, а затем порядковый номер элемента в квадратных скобках (`[]`). Порядковый номер элемента программисты называют **индексом (index)**. Первый элемент любого массива имеет индекс `0` (ноль). Таким образом, массив `c` содержит элементы от `c[0]` до `c[4]`. Значение `c[0]` равно `-45`, значение `c[2]` равно `0`, а значение `c[4]` равно `1543`.

Последний индекс (в данном случае `4`) всегда на единицу меньше, чем количество элементов массива (в данном случае `5`). Каждый массив «знает» свой размер, который вы можете получить с помощью функции `size`:

```
c.size()
```

Индекс должен быть целым числом или целочисленным выражением. Квадратные скобки, содержащие индекс, — это **оператор индекса (subscript operator)**. Он имеет тот же приоритет, что и круглые скобки в вызове функции. Результатом операции с оператором индекса является значение *lvalue* — его можно использовать в левой части присваивания, так же как и имя переменной. Например, следующая инструкция присваивает элементу `c[4]` новое значение `87`:

```
c[4] = 87;
```

Полная таблица приоритетов операторов приведена в приложении А.

6.3. Объявление массива

Чтобы объявить массив, надо указать его тип данных, размер (количество элементов этого типа) и имя:

```
std::array<тип, размерМассива> имяМассива;
```

Запись в угловых скобках <*тип*, *размерМассива*> показывает, что `array` — это **шаблон класса (class template)**. Подобно шаблону функции, шаблон класса

используется компилятором для создания множества **специализаций шаблона класса (class template specializations)**, работающих с разными типами данных. Например, специализациями одного и того же шаблона могут быть: массив из 5 элементов типа `int`; массив из 7 элементов типа `double`; массив из 100 объектов пользовательского типа `Employee` (каждый из которых хранит сведения о сотруднике предприятия). В главе 9 вы узнаете, как создавать пользовательские типы данных. Компилятор резервирует необходимый объем памяти в зависимости от типа и размера массива. Например, следующее объявление выделит память для массива с именем `c`, состоящего из 5 целых чисел:

```
std::array<int, 5> c; // c - это массив из 5 целых чисел
```

6.4. Инициализация массива в цикле

В программе, показанной на илл. 6.1, объявляется массив с именем `values` (строка 8), состоящий из пяти целых чисел. В строке 5 включен заголовок `<array>`, содержащий определение шаблона класса `array`.

```
 1 // fig06_01.cpp
 2 // Инициализация массива и вывод на экран
 3 #include <fmt/format.h> // Или в C++20: #include <format>
 4 #include <iostream>
 5 #include <array>
 6
 7 int main() {
 8    std::array<int, 5> values; // values - это массив из 5 целых чисел
 9
10    // Инициализируем все элементы массива нулями
11    for (size_t i{0}; i < values.size(); ++i) {
12       values[i] = 0; // Присваиваем значение 0 элементу с индексом i
13    }
14
15    std::cout << fmt::format("{:>7}{:>10}\n", "Element", "Value");
16
17    // Выводим значения всех элементов массива
18    for (size_t i{0}; i < values.size(); ++i) {
19       std::cout << fmt::format("{:>7}{:>10}\n", i, values[i]);
20    }
21
22    std::cout << fmt::format("\n{:>7}{:>10}\n", "Element", "Value");
23
24    // Обращаемся к элементам через функцию at
25    for (size_t i{0}; i < values.size(); ++i) {
26       std::cout << fmt::format("{:>7}{:>10}\n", i, values.at(i));
27    }
28
29    // Пробуем обратиться к несуществующему элементу с индексом 10
30    values.at(10); // Функция at выдает исключение
31 }
```

```
Element     Value
      0         0
      1         0
      2         0
      3         0
      4         0

Element     Value
      0         0
      1         0
      2         0
      3         0
      4         0
terminate called after throwing an instance of
what(): array::at: __n (which is 10) >= _Nm (which is 5)
Aborted
```

Илл. 6.1 | Инициализация массива и вывод на экран

Присваивание значений всем элементам

Строки 11–13 — это цикл `for`, в котором каждому элементу массива присваивается значение `0`. Элементы массивов, как и обычные нестатические локальные переменные, не инициализируются компилятором автоматически (неявно). Исключением из этого правила являются статические (`static`) массивы, которые компилятор автоматически (неявно) инициализирует нулями. Выполняя в цикле последовательную обработку элементов массива, в том числе для их инициализации, программист должен убедиться, что условие продолжения цикла исключает опасность обращения к несуществующим элементам за пределами массива. В разделе 6.6 мы представим вам цикл `for` для диапазона значений, который не имеет условия продолжения и поэтому позволяет выполнять обработку всех элементов массива более безопасно.

Тип size_t

В строках 11, 18 и 25 управляющие переменные каждого цикла объявляются с типом `size_t`. Согласно стандарту C++, `size_t` — это целочисленный тип без знака, предназначенный для обозначения размеров любых объектов[1]. Используйте переменные этого типа, чтобы обозначать размеры и индексы массивов. Тип `size_t` находится в пространстве имен `std` и определен в заголовке `<cstddef>`, который автоматически включается другими заголовками стандартной библиотеки, использующими `size_t`. Если компилятор «не знает» тип `size_t` и сообщает об ошибке, добавьте в вашу программу директиву `#include <cstddef>`.

[1] Стандарт C++, «17.2.4. Sizes, Alignments, and Offsets». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/support.types#layout-3.

Отображение элементов массива

Первая инструкция вывода (строка 15) отображает заголовки столбцов таблицы. Затем в цикле `for` выводятся значения элементов массива (строки 18–20). Чтобы отобразить данные в виде таблицы, мы включили в инструкции вывода появившиеся в C++20 средства форматирования текста, упомянутые в разделе 4.14.

Как избежать ошибок: проверка границ массива

Если программист обращается к элементу массива, используя квадратные скобки (как в строках 11–13 и 18–20), компилятор C++ не выполняет **проверку границ (bounds checking)** массива, которая могла бы предотвратить обращение к несуществующему элементу. Из-за этого выполняемые программы могут «вылетать» за границы массивов без предупреждения. В отличие от квадратных скобок, функция `at` из шаблона класса `array` выполняет проверку границ массива. В строках 25–27 демонстрируется обращение к элементам через функцию `at`. Вы также можете присваивать значения элементам массива, используя функцию `at` в левой части инструкции присваивания:

```
values.at(0) = 10;
```

В строке 30 мы пробуем обратиться к несуществующему элементу массива. Когда функция `at`, проверив полученный индекс, обнаруживает выход за границы массива, она генерирует **ошибку выполнения (runtime error)**, которая называется **исключением (exception)**. Поэтому в строке 30 программа, скомпилированная с помощью GNU g++ и запущенная в Linux, завершается с сообщением об ошибке:

```
terminate called after throwing an instance of
  what(): array::at: __n (which is 10) >= _Nm (which is 5)
Aborted
```

Это сообщение означает, что функция `at` (`array::at`) выяснила, что переменная `__n` (со значением `10`) больше или равна переменной `_Nm` (со значением `5`). В GNU C++ функция `at` реализована так, что ее переменная `__n` представляет индекс элемента, а ее переменная `_Nm` представляет размер массива. То есть функция проверила границы массива, обнаружила недопустимый индекс и выдала исключение. В разделе 6.15 мы покажем, как обработать исключение для корректной реакции на ошибку во время выполнения программы. В главе 12 обработка исключений рассматривается подробнее.

Программы, в которых возможны инструкции чтения и записи значений вне границ массивов, часто несут в себе опасность. Попытка чтения несуществующего элемента массива может привести к сбою программы или даже к подмене данных при внешне нормальной работе программы. Попытка записи в несуществующий элемент (переполнение буфера[1]) может привести к повреждению данных в па-

[1] Дополнительные сведения о переполнении буфера: https://ru.wikipedia.org/wiki/Переполнение_буфера.

мяти и к краху программы. В некоторых случаях переполнением буфера могут пользоваться злоумышленники, заменяя часть исполняемого кода программы вредоносным кодом.

6.5. Инициализация массива со списком инициализаторов

Все элементы массива можно инициализировать непосредственно в инструкции, которая объявляет массив. Для этого надо указать после имени массива разделенный запятыми список **инициализаторов (initializers)** в фигурных скобках. В программе, показанной на илл. 6.2, есть два **списка инициализаторов (initializer list)**: для массива с пятью элементами `int` (строка 8) и для массива с четырьмя элементами `double` (строка 18). Значения обоих массивов выводятся на экран.

- В строке 8 явно указаны тип элементов (`int`) и размер массива (`5`).
- В строке 18 компилятор автоматически определяет тип элементов (`double`) и размер массива (`4`) по списку инициализаторов. Этот механизм, появившийся в C++17, называется **дедуктивным определением аргументов шаблона класса (class template argument deduction**, или сокращенно **CTAD)**[1]. Когда вы заранее знаете начальные значения элементов контейнера, CTAD может упростить ваш код.

```
1 // fig06_02.cpp
2 // Инициализация массива при объявлении
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <array>
6
7 int main() {
8    std::array<int, 5> values{32, 27, 64, 18, 95}; // Скобочная инициализация
9
10   // Выводим на экран значение каждого элемента
11   for (size_t i{0}; i < values.size(); ++i) {
12      std::cout << fmt::format("{} ", values.at(i));
13   }
14
15   std::cout << "\n\n";
16
17   // Объявляем массив в краткой форме записи, используя CTAD
18   std::array values2{1.1, 2.2, 3.3, 4.4};
19
20   // Выводим на экран значение каждого элемента
21   for (size_t i{0}; i < values.size(); ++i) {
```

Окончание ⇨

[1] «Class Template Argument Deduction (CTAD)». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/language/class_template_argument_deduction.

```
22        std::cout << fmt::format("{} ", values.at(i));
23    }
24
25    std::cout << "\n";
26 }
```

```
32 27 64 18 95
1.1 2.2 3.3 4.4
```

Илл. 6.2 | Инициализация массива при объявлении

Инициализация по умолчанию

Если количество инициализаторов в списке меньше, чем количество элементов массива, все остальные элементы **инициализируются значением по умолчанию (value initialized)**: для основных числовых типов — `0`, для типа `bool` — `false`, а объекты инициализируются в соответствии с определениями их классов (мы расскажем об этом в главе 9). Например, следующее объявление инициализирует все элементы массива `int` нулями, так как в списке нет ни одного инициализатора (ноль), что меньше количества элементов массива:

```
std::array<int, 5> values{}; // Инициализация значением 0
```

Этот удобный метод можно использовать и для повторной инициализации элементов существующего массива во время выполнения программы:

```
values = {}; // Обнуление всех элементов массива
```

Чрезмерно длинный список инициализаторов

Если при объявлении массива явно указан его размер, количество инициализаторов в списке должно быть не больше этого размера. Например, следующее объявление вызовет ошибку компиляции, так как в нем перечислены шесть инициализаторов для массива из пяти элементов:

```
std::array<int, 5> values{32, 27, 64, 18, 95, 14};
```

6.6. Цикл for для диапазона значений (C++11) и цикл for для диапазона значений с инициализатором (C++20)

В большинстве случаев при обработке массива должны обрабатываться все его элементы, от начала до конца. **Инструкция `for` для диапазона значений (range-based `for` statement)**, появившаяся в C++11, позволяет делать это без счетчика. Как правило, для обработки массивов предпочтителен именно этот вид цикла, поскольку он не позволяет программам «вылетать» за границы массивов. В конце этого раздела мы сравним два вида цикла: `for` со счетчиком (с управляющей переменной) и `for` для диапазона значений. В программе, показанной на илл. 6.3,

инструкции for для диапазона значений используются для вывода на экран элементов массива (строки 12–14, 23–25) и для математической обработки массива (строки 17–19).

```
1 // fig06_03.cpp
2 // Цикл for для диапазона значений
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <array>
6
7 int main() {
8    std::array items{1, 2, 3, 4, 5}; // Автоопределение типа: array<int, 5>
9
10   // Выводим на экран массив до изменения
11   std::cout << "items before modification: ";
12   for (const int& item : items) { // item - константная ссылка на int
13      std::cout << fmt::format("{} ", item);
14   }
15
16   // Удваиваем все элементы массива
17   for (int& item : items) { // item - ссылка на элемент типа int
18      item *= 2;
19   }
20
21   // Выводим на экран массив после изменения
22   std::cout << "\nitems after modification: ";
23   for (const int& item : items) {
24      std::cout << fmt::format("{} ", item);
25   }
26
27   // Суммируем элементы в цикле for для диапазона значений
28   std::cout << "\n\ncalculating a running total of items' values:\n";
29   for (int runningTotal{0}; const int& item : items) {
30      runningTotal += item;
31      std::cout << fmt::format("item: {}; running total: {}\n",
32         item, runningTotal);
33   }
34 }
```

```
items before modification: 1 2 3 4 5
items after modification: 2 4 6 8 10

calculating a running total of items
item: 2; running total: 2
item: 4; running total: 6
item: 6; running total: 12
item: 8; running total: 20
item: 10; running total: 30
```

Илл. 6.3 | Цикл for для диапазона значений

Отображение элементов массива в цикле for для диапазона значений

Цикл `for` для диапазона значений упрощает код, выполняющий последовательное обращение ко всем элементам массива. Строку 12 можно прочитать так: «Для каждого элемента `item` в массиве `items` следует выполнить...». В каждой итерации цикла эта инструкция присваивает переменной `item` значение очередного элемента массива `items`, а затем выполняет тело цикла. Обратите внимание, что в каждой итерации цикла переменная `item` ссылается на значение (а не на индекс!) элемента массива. Эта переменная, объявленная в заголовке цикла перед двоеточием, называется **переменной диапазона (range variable)**. После двоеточия в заголовке цикла указано имя массива[1]. В нашем примере переменная `item` объявлена как константная ссылка (с квалификатором `const`). Напомним, что ссылка — это псевдоним для другой переменной в памяти, в данном случае для одного из элементов массива.

Если переменная диапазона является константной ссылкой, цикл выполняется быстрее. Ссылка не позволяет циклу копировать каждый элемент массива в переменную диапазона. Эта экономия времени особенно важна при работе с большими объектами.

Обработка элементов массива в цикле for для диапазона значений

В строках 17–19 цикл `for` для диапазона значений используется для умножения на 2 каждого элемента массива `items`. В строке 17 переменная `item` объявлена как ссылка на значение типа `int` (`int&`). Эта ссылка не константная (без квалификатора `const`), поэтому при каждом изменении переменной `item` меняется значение соответствующего элемента массива.

Цикл for для диапазона значений с инициализатором (C++20)

В C++20 появилась еще одна разновидность цикла `for` для диапазона значений: с инициализатором в заголовке цикла. Как и в инструкциях `if` и `switch` с инициализаторами, переменные, определенные в заголовке цикла `for` для диапазона значений, существуют только во время выполнения цикла. В строке 29 переменная `runningTotal` инициализируется значением `0`, затем в строках 29–33 суммируются элементы массива `items`. После завершения цикла переменная `runningTotal` выходит из области видимости, то есть перестает существовать.

Обращение к элементам массива без индексов

Во многих случаях традиционный цикл `for` со счетчиком можно заменить циклом `for` для диапазона значений. Например, для суммирования всех целых чисел в массиве нужен доступ только к значениям элементов, а их индексные позиции в массиве не важны.

---

[1] Инструкцию for для диапазона значений можно использовать с большинством контейнеров стандартной библиотеки C++, которые обсуждаются в главе 13.

Два вида итерации: внешняя и внутренняя

В рассмотренной программе строки 12–14 можно было бы заменить функционально эквивалентной инструкцией цикла `for` со счетчиком:

```
for (int counter{0}; counter < items.size(); ++counter) {
   std::cout << fmt::format("{} ", items[counter]);
}
```

Но этот стиль итерации, известный как **внешняя итерация (external iteration)**, не вполне надежен. Дело в том, что для правильной обработки массива ваш код должен многократно изменять значение управляющей переменной (счетчика). Каждый раз, когда вы пишете подобную инструкцию, вы рискуете допустить ошибку, например:

- неправильно инициализировать управляющую переменную;
- неправильно задать условие продолжения цикла;
- неправильно задать инкремент управляющей переменной.

Каждая из этих ошибок может привести к попытке доступа к несуществующим элементам массива за пределами его границ.

Инструкция `for` для диапазона значений использует **внутреннюю итерацию (internal iteration)**, скрывая от вас все детали. Вы просто указываете, какой массив нужно обработать. Инструкция знает, как получить каждое значение из массива, и завершит итерацию, когда значения закончатся.

6.7. Заполнение массива и спецификатор constexpr

На илл. 6.4 показано, как массив `values` заполняется четными целыми числами `2`, `4`, `6`, `8` и `10` (строки 13–15), которые затем выводятся на экран (строки 18–20). Каждое число генерируется в строке 14 путем умножения управляющей переменной `i` на `2` и прибавления `2`.

```
 1 // fig06_04.cpp
 2 // Заполнение массива четными числами от 2 до 10
 3 #include <fmt/format.h>
 4 #include <iostream>
 5 #include <array>
 6
 7 int main() {
 8    // Константа подходит для указания размера массива
 9    constexpr size_t arraySize{5}; // Надо инициализировать в объявлении
10
11    std::array<int, arraySize> values{}; // В массиве values 5 элементов
12
13    for (size_t i{0}; i < values.size(); ++i) { // Заполняем массив
14       values.at(i) = 2 + 2 * i;
15    }
```

Окончание ⇨

```
16
17    // Выводим значения элементов массива в виде таблицы
18    for (const int& value : values) {
19       std::cout << fmt::format("{} ", value);
20    }
21
22    std::cout << '\n';
23 }
```

```
2 4 6 8 10
```

Илл. 6.4 | Заполнение массива четными числами от 2 до 10

Константы

В главе 5 вы познакомились с квалификатором `const`, запрещающим изменять значение переменной после ее инициализации. В C++11 появился спецификатор `constexpr`. Он означает, что компилятор должен вычислить значение переменной при компиляции и затем сделать эту переменную константой. Это повышает быстродействие, поскольку в выполняемой программе не тратится время на инициализацию. Переменная, объявленная со спецификатором `constexpr`, — это тоже константа, но с очень важным для компилятора отличием: она должна быть инициализирована во время компиляции. Обычные `const`-переменные вы можете инициализировать во время выполнения.

В строке 9 спецификатор `constexpr` используется для объявления константы `arraySize` со значением `5`. Переменная, объявленная как `constexpr` или `const`, должна быть инициализирована при ее объявлении; иначе возникнет ошибка компиляции. Например, ошибку компиляции вызовет попытка изменить размер массива после его инициализации[1]:

```
arraySize = 7;
```

Если в определении массива вы указываете его размер в виде константы (например, `arraySize` вместо литерала `5`), ваш код становится понятнее и проще для последующей модификации. Этот метод устраняет **магические числа (magic numbers)** — встречающиеся в коде числовые литералы, смысл которых трудно понять. Заменяя литералы константами, вы можете тем самым пояснять их смысл, давая им понятные имена.

Вызовы constexpr-функций вычисляются во время компиляции

Одна из современных тенденций языка C++ заключается в том, чтобы выполнять как можно больше полезных вычислений на этапе компиляции. Это нужно и для надежной проверки правильности типов данных в исходном коде, и для ускорения работы программ. Каждый новый стандарт C++, начиная с C++11, расширял

[1] Некоторые компиляторы в сообщениях об ошибках называют переменные-константы основных типов «константными объектами» («const objects»). Согласно стандарту C++, объект — это «область памяти» («region of storage»). Подобно объектам классов, переменные основных типов тоже занимают место в памяти, и в этом смысле о них часто говорят как об объектах.

возможности спецификатора `constexpr`. В последующих главах вы увидите, что его теперь можно применять даже к функциям. Если в исходном коде есть вызов `constexpr`-функции и все передаваемые ей аргументы тоже обозначены как `constexpr`, компилятор может заранее вычислить результат этого вызова, получив константу. Это упрощает исполняемый код и повышает быстродействие. В C++20, C++17 и C++14 многие функции стандартной библиотеки C++ объявлены как `constexpr`.

Подробную информацию о спецификаторе `constexpr` и о способах его применения вы найдете здесь:

https://en.cppreference.com/w/cpp/language/constexpr

В главе 15 мы расскажем о метапрограммировании на основе шаблонов, в результате которого часть вычислений переносится с этапа выполнения программы на этап компиляции.

6.8. Суммирование элементов массива

Часто элементы массива представляют собой исходные данные для вычислений. Например, если в массив занесены экзаменационные оценки, преподаватель может захотеть суммировать их, а затем вычислить среднюю арифметическую оценку по классу. Преобразование множества значений в одно результирующее значение называется **сверткой (reduction)**. Это одна из ключевых операций в функциональном программировании, мы обсудим ее в разделе 6.14. На илл. 6.5 показан пример суммирования элементов массива в цикле `for` для диапазона значений (строки 12–14). В разделе 6.14 мы выполним это же вычисление с помощью функции `accumulate` стандартной библиотеки C++.

```
1 // fig06_05.cpp
2 // Суммирование элементов массива
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <array>
6
7 int main() {
8    std::array items{10, 20, 30, 40}; // Автоопределение типа: array<int, 4>
9    int total{0};
10
11    // Суммируем элементы
12    for (const int& item : items) {
13       total += item;
14    }
15
16    std::cout << fmt::format("Total of array elements: {}\n", total);
17 }
```

```
Total of array elements: 100
```

Илл. 6.5 | Суммирование элементов массива

6.9. Визуализация данных массива на простой диаграмме

Многие программы отображают данные графически. Например, множество чисел можно показать на гистограмме (столбчатой диаграмме), высота столбцов которой пропорциональна величине чисел. Один из самых простых способов визуализации чисел — вывести на экран диаграмму, отображающую каждое число в виде ряда звездочек (`*`).

Преподаватель может визуализировать результаты экзамена, распределив все оценки по нескольким категориям, или диапазонам. Предположим, что оценки учеников таковы: `87`, `68`, `94`, `100`, `83`, `78`, `85`, `91`, `76` и `87`. Только один ученик получил максимальную оценку `100`, два ученика — оценки в диапазоне от `90` до `99`, четыре — от `80` до `89`, два — от `70` до `79`, один — от `60` до `69` и ни одного — ниже `60`. Наша следующая программа (илл. 6.6) хранит эти данные в массиве `frequencies`, состоящем из `11` элементов, каждый из которых соответствует своему диапазону. Например, элемент с индексом `0` содержит количество оценок в диапазоне от `0` до `9`, элемент с индексом `7` — количество оценок в диапазоне от `70` до `79`, а элемент с индексом `10` — количество оценок `100`. В разделах 6.10 и 6.11 мы покажем примеры кода, который вычисляет частотное распределение в наборе данных (то есть подсчитывает, как часто случалось каждое из нескольких возможных событий). А массив `frequencies` из примера кода в этом разделе мы инициализировали готовыми данными о частотном распределении.

Программа (илл. 6.6) считывает элементы массива `frequencies` и отображает их в виде диаграммы. Каждая строка вывода начинается с заголовка, за которым следует ряд звездочек, обозначающий количество оценок в этом диапазоне. Программа работает так:

- В строке 8 массив `frequencies` объявлен со спецификатором `constexpr`, потому что программа не должна изменять элементы массива, известные во время компиляции.
- Заголовок каждой строки диаграммы (например, `70-79:`) формируется в строках 15–21 на основе значения переменной `i`. В строках 16–17 спецификатор формата `:02d` указывает, что целое число (`d`) должно быть отформатировано в поле из `2` знаков, причем с добавлением начальной цифры `0` в том случае, если оно состоит менее чем из двух цифр. В строке 20 спецификатор формата `:>5d` указывает, что число должно быть выровнено по правому краю (`>`) в поле шириной `5` знаков.
- Вложенный цикл `for` (строки 26–28) выводит звездочки в текущей строке. Условие продолжения в строке 26 (`stars < frequency`) задано так, что количество итераций этого цикла равно `frequency`. Следовательно, значение каждого из элементов массива `frequencies` определяет количество звездочек в соответствующей строке диаграммы. Элементы массива с индексами от `0` до `5` содержат нули, потому что никто не получил оценку ниже `60` бал-

лов. Таким образом, в первых шести строках диаграммы звездочки не отображаются.

```
// fig06_06.cpp
// Частотное распределение результатов экзамена на простой диаграмме
#include <fmt/format.h>
#include <iostream>
#include <array>

int main() {
   constexpr std::array frequencies{0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1};

   std::cout << "Grade distribution:\n";

   // Для каждого элемента frequencies выводим строку диаграммы
   for (int i{0}; const int& frequency : frequencies) {
      // Выводим заголовки диаграммы ("00-09:", ..., "90-99:", "100:")
      if (i < 10) {
         std::cout << fmt::format("{:02d}-{:02d}: ",
            i * 10, (i * 10) + 9);
      }
      else {
         std::cout << fmt::format("{:>5d}: ", 100);
      }

      ++i;

      // Выводим ряд звездочек
      for (int stars{0}; stars < frequency; ++stars) {
         std::cout << '*';
      }

      std::cout << '\n'; // Перенос строки - подготовка к следующей строке
   }
}
```

```
Grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

Илл. 6.6 | Частотное распределение результатов экзамена на простой диаграмме

6.10. Использование элементов массива в качестве счетчиков

Иногда программы используют переменные-счетчики для обобщения данных, например результатов опроса. Когда мы моделировали 60 миллионов бросков кубика (см. илл. 5.3), у нас было шесть отдельных счетчиков, каждый из которых увеличивался при выпадении соответствующей грани кубика. На илл. 6.7 показано аналогичное моделирование, но на этот раз мы сделали счетчики элементами массива.

```
1 // fig06_07.cpp
2 // Моделирование бросков кубика с помощью массива (вместо switch)
3 #include <fmt/format.h> // В C++20:  #include <format>
4 #include <iostream>
5 #include <array>
6 #include <random>
7
8 int main() {
9    // Инициализируем генератор случайных чисел
10   std::random_device rd; // Инициализатор default_random_engine
11   std::default_random_engine engine{rd()}; // rd() инициализирует генератор
12   std::uniform_int_distribution randomDie{1, 6};
13
14   constexpr size_t arraySize{7}; // Игнорируем элемент с индексом 0
15   std::array<int, arraySize> frequency{}; // Инициализируем массив нулями
16
17   // Бросаем кубик 60 миллионов раз; индекс массива - случайное число
18   for (int roll{1}; roll <= 60'000'000; ++roll) {
19      ++frequency.at(randomDie(engine));
20   }
21
22   std::cout << fmt::format("{}{:>13}\n", "Face", "Frequency");
23
24   // Выводим все элементы массива
25   for (size_t face{1}; face < frequency.size(); ++face) {
26      std::cout << fmt::format("{:>4}{:>13}\n", face, frequency.at(face));
27   }
28 }
```

```
Face    Frequency
   1      9997901
   2      9999110
   3     10001172
   4     10003619
   5      9997606
   6     10000592
```

Илл. 6.7 | Моделирование бросков кубика с помощью массива (вместо switch)

В этой программе результаты бросков кубика подсчитываются в массиве `frequency` (строка 15). Инструкция в строке 19 заменяет инструкцию `switch` из строк 23–45

прежней версии программы (илл. 5.3). В каждом броске в качестве индекса выбирается случайное число, определяющее индекс того элемента массива `frequency`, который надо изменить. Оператор «точка» (`.`) в строке 19 имеет более высокий приоритет, чем оператор `++`. Таким образом, эта инструкция выбирает случайный элемент массива и увеличивает его значение на `1`.

Вызов `randomDie(engine)` возвращает случайное число от `1` до `6`, поэтому массив `frequency` должен хранить не менее шести элементов. Мы решили сделать массив из семи элементов и игнорировать элемент с нулевым индексом, чтобы сделать код более понятным, ведь при выпадении числа `1` обращение к элементу `frequency.at(1)` выглядит более наглядно, чем обращение к `frequency.at(0)`. Каждое выпавшее число используется непосредственно в качестве индекса массива `frequency`. Мы также заменили строки 49–54 из прежней версии программы (илл. 5.3): теперь результаты выводятся с помощью цикла (илл. 6.7, строки 25–27). Чтобы пропустить при выводе элемент с индексом `0`, счетчик этого цикла инициализирован значением `1`.

6.11. Использование массива для подсчета результатов опроса

В нашем следующем примере массивы используются для обобщения данных, собранных в ходе опроса. Рассмотрим следующую постановку задачи:

> *Двадцати студентам предложили оценить качество еды в студенческой столовой по шкале от 1 до 5, где 1 — «ужасно», а 5 — «превосходно». Поместите 20 ответов в массив чисел типа int и вычислите частотное распределение ответов.*

Этот распространенный вид задач сводится к обработке массива (илл. 6.8). Мы хотим подсчитать количество каждого из пяти возможных значений (от `1` до `5`). Массив `responses` (строки 9–10) содержит `20` элементов, инициализированных ответами студентов. Он объявлен как `constexpr`, потому что его элементы не должны изменяться и известны во время компиляции. Массив `frequency` (строка 18) содержит `6` элементов, он нужен для подсчета количества ответов. Каждый элемент массива `frequency` инициализируется числом `0` и затем служит счетчиком для одного из возможных значений. Как и в предыдущей программе (илл. 6.7), элемент с индексом `0` игнорируется.

Первая инструкция `for` (строки 18–20) получает в каждой итерации цикла очередной ответ студента (число от `1` до `5`) из массива `responses` и увеличивает один из счетчиков: от `frequency.at(1)` до `frequency.at(5)`. Ключевая инструкция — это строка 19, в которой выбор инкрементируемого счетчика зависит от значения `responses.at(response)`. Независимо от количества ответов, программе нужен для подсчета результатов массив только из шести элементов (элемент с индексом `0` игнорируется), поскольку все возможные ответы находятся в диапазоне от `1` до `5`, а массив из шести элементов имеет индексы от `0` до `5`.

```
// fig06_08.cpp
// Программа для подсчета результатов опроса
#include <fmt/format.h> // В C++20 - #include <format>
#include <iostream>
#include <array>

int main() {
   // Результаты опроса уже занесены в массив responses
   constexpr std::array responses{
      1, 2, 5, 4, 3, 5, 2, 1, 3, 1, 4, 3, 3, 3, 2, 3, 3, 2, 2, 5};

   // Инициализируем массив frequency нулями
   constexpr size_t frequencySize{6}; // Размер массива
   std::array<int, frequencySize> frequency{};

   // Используем число из массива responses в качестве
   // индекса массива frequency для инкремента соответствующего счетчика
   for (const int& response : responses) {
      ++frequency.at(response);
   }

   std::cout << fmt::format("{}{:>12}\n", "Rating", "Frequency");

   // Выводим результаты на экран
   for (size_t rating{1}; rating < frequency.size(); ++rating) {
      std::cout << fmt::format("{:>6}{:>12}\n",
         rating, frequency.at(rating));
   }
}
```

```
Rating   Frequency
     1           3
     2           5
     3           7
     4           2
     5           3
```

Илл. 6.8 | Программа подсчета результатов опроса

6.12. Сортировка массива и поиск в массиве

В этом разделе мы продемонстрируем две функции стандартной библиотеки C++. Функция `sort`[1] упорядочит элементы нашего массива в порядке возрастания, а функция `binary_search` выполнит в нем поиск заданных элементов.

Сортировка данных (data sorting), то есть их размещение в порядке возрастания или убывания, относится к важнейшим задачам программирования. Почти каждой организации нужно сортировать какие-либо данные, причем во многих

[1] Стандарт C++ указывает, что сортировка использует алгоритм $O(n \log n)$, но не уточняет, какой именно.

случаях их объемы очень велики. Сортировка — это интереснейшая проблема, для решения которой математики и программисты выполнили много сложных исследовательских работ.

При обработке массива часто требуется выяснить, совпадает ли значение какого-либо его элемента с **ключевым значением (key value)**. Процесс сверки элементов массива с ключевым значением называется **поиском (searching)**.

Функции sort и binary_search

В нашей следующей программе (илл. 6.9) создается неотсортированный массив `colors` из семи объектов типа `string` (строки 13–14). Объявление `using` в строке 10 позволяет инициализировать этот массив **объектными литералами (object literals)** типа `string`. Такой литерал обозначается символом `s` после закрывающей кавычки. Например, если вы напишете `"red"s`, символ `s` сообщит компилятору, что этот литерал является объектом класса `std::string`. По инициализаторам в строках 13–14 компилятор определяет, что элементы этого массива должны иметь тип `std::string`. Цикл в строках 18–20 выводит на экран содержимое массива `colors`. Обратите внимание, что переменная диапазона объявлена как константная ссылка:

```
const std::string&
```

Благодаря ссылке, в этом цикле не выполняется поочередное копирование каждого объекта типа `string` в переменную `color`. В программах, обрабатывающих большое количество объектов типа `string`, такое копирование существенно снижает быстродействие.

```
 1 // fig06_09.cpp
 2 // Сортировка массива и поиск в массиве
 3 #include <array>
 4 #include <algorithm> // Содержит функции sort и binary_search
 5 #include <fmt/format.h>
 6 #include <iostream>
 7 #include <string>
 8
 9 int main() {
10    using namespace std::string_literals; // Строковые объектные литералы
11
12    // Компилятор автоматически определяет, что colors - это array<string, 7>
13    std::array colors{"red"s, "orange"s, "yellow"s,
14       "green"s, "blue"s, "indigo"s, "violet"s};
15
16    // Выводим на экран массив до сортировки
17    std::cout << "Unsorted colors array:\n ";
18    for (const std::string& color : colors) {
19       std::cout << fmt::format("{} ", color);
20    }
21
22    // Сортируем массив
23    std::sort(std::begin(colors), std::end(colors));
```

Окончание ⇨

```
24
25    // Выводим на экран отсортированный массив
26    std::cout << "\nSorted colors array:\n ";
27    for (const std::string& color : colors) {
28       std::cout << fmt::format("{} ", color);
29    }
30
31    // Есть ли в массиве элемент "indigo"?
32    bool found{std::binary_search(
33       std::begin(colors), std::end(colors), "indigo")};
34    std::cout << fmt::format("\n\n\"indigo\" {} found in colors array\n",
35       found ? "was" : "was not");
36
37    // Есть ли в массиве элемент "cyan"?
38    found = std::binary_search(
39       std::begin(colors), std::end(colors), "cyan");
40    std::cout << fmt::format("\"cyan\" {} found in colors array\n",
41       found ? "was" : "was not");
42 }
```

```
Unsorted colors array:
   red orange yellow green blue indigo violet
Sorted colors array:
   blue green indigo orange red violet yellow

"indigo" was found in colors array
"cyan" was not found in colors array
```

Илл. 6.9 | Сортировка массива и поиск в массиве

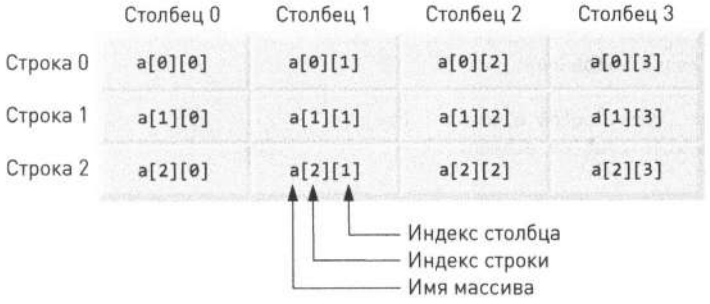
В строке 23 вызывается функция `sort` (заголовок `<algorithm>` стандартной библиотеки C++), которая сортирует массив `colors` в порядке возрастания. Для объектов типа `string` эта функция выполняет **лексикографическую сортировку (lexicographical sorting)**, то есть упорядочивает текстовые строки по числовым значениям их символов в соответствующей кодировке. Аргументы функции `sort` определяют диапазон сортируемых элементов, в данном случае это весь массив. Аргументы `std::begin(colors)` и `std::end(colors)` возвращают **итераторы (iterators)**, представляющие начало и конец массива соответственно. Итераторы подробно рассматриваются в главе 13. Функции `begin` и `end` определены в заголовке `<array>`. Как вы увидите в последующих главах, функция `sort` может сортировать элементы в структурах данных нескольких типов. Цикл в строках 27–29 выводит на экран содержимое отсортированного массива.

В строках 32–33 и 38–39 вызывается функция `binary_search` (заголовок `<algorithm>` стандартной библиотеки C++), которая определяет, есть ли среди элементов массива определенное значение. Перед вызовом `binary_search` любой массив обязательно должен быть отсортирован в порядке возрастания, сама функция этого не делает. Использование этой функции для поиска в неотсортированном массиве является логической ошибкой, приводящей к неправиль-

ным результатам. Первые два аргумента функции `binary_search` представляют диапазон элементов массива, среди которых выполняется поиск, а третий — это ключевое значение, которое надо найти. Функция возвращает результат типа `bool`, указывающий, было ли найдено ключевое значение. В главе 14 мы будем использовать стандартную функцию C++ `find`, возвращающую индекс ключевого значения в массиве.

6.13. Многомерные массивы

Вы можете использовать массивы с двумя измерениями (то есть с двумя индексами) для представления **таблиц значений (tables of values)**, в которых данные расположены по **строкам (rows)** и **столбцам (columns)**. Чтобы идентифицировать конкретный элемент таблицы, надо указать два индекса. По соглашению, первый индекс идентифицирует строку, а второй — столбец. Массивы, в которых элементы идентифицируются двумя индексами, называются **двумерными (two-dimensional)** или **2D-массивами**. Массивы с двумя и более измерениями называются **многомерными (multidimensional)**. На диаграмме показан двумерный массив с именем a:



В этом массиве три строки и четыре столбца, поэтому его называют «массив `3` на `4`». В общем случае о массиве, в котором `m` строк и `n` столбцов, говорят, что это «массив `m` на `n`».

На диаграмме мы идентифицировали каждый элемент по имени в виде `a`[*строка*][*столбец*]. Доступ к каждому элементу также можно получить с помощью функции `at`:

```
a.at(i).at(j)
```

Все имена элементов первой строки (строки `0`) имеют первый индекс `0`; все имена элементов последнего столбца (столбца `3`) имеют второй индекс `3`.

В программе, показанной на илл. 6.10, два двумерных массива инициализируются в объявлениях. Каждая из строк 11–12 создает двумерный массив (то есть массив массивов) с двумя строками и тремя столбцами.

```
// fig06_10.cpp
// Инициализация многомерных массивов
#include <iostream>
#include <array>

constexpr size_t rows{2};
constexpr size_t columns{3};
void printArray(const std::array<std::array<int, columns>, rows>& a);

int main() {
   constexpr std::array values1{std::array{1, 2, 3}, std::array{4, 5, 6}};
   constexpr std::array values2{std::array{1, 2, 3}, std::array{4, 5, 0}};

   std::cout << "values1 by row:\n";
   printArray(values1);

   std::cout << "\nvalues2 by row:\n";
   printArray(values2);
}

// Выводим массив с двумя строками и тремя столбцами
void printArray(const std::array<std::array<int, columns>, rows>& a) {
   // Обрабатываем строки массива в цикле
   for (const auto& row : a) {
      // Обрабатываем столбцы текущей строки массива в цикле
      for (const auto& element : row) {
         std::cout << element << ' ';
      }

      std::cout << '\n'; // Новая строка для вывода
   }
}
```

```
values1 by row:
1 2 3
4 5 6

values2 by row:
1 2 3
4 5 0
```

Илл. 6.10 | Инициализация многомерных массивов

Объявление двумерного массива

В строках 11–12 компилятор автоматически определяет, что `values1` и `values2` — это двумерные массивы элементов типа `int`, расположенных в две строки по три столбца в каждой. Рассмотрим два инициализатора массива `values1`:

```
std::array{1, 2, 3}
std::array{4, 5, 6}
```

По ним компилятор определяет, что `values1` имеет две строки. Каждый инициализатор создает массив из трех элементов, поэтому компилятор определяет, что каждая строка `values1` имеет три столбца. Наконец, значения в этих инициализаторах являются целыми числами (`int`), поэтому компилятор определяет, что элементы `values1` имеют тип `int`.

Вывод на экран элементов двумерного массива

Для вывода элементов `values1` и `values2` программа вызывает функцию `printArray`. Прототип функции (строка 8) и ее определение (строки 22–32) указывают, что `printArray` получает через константную ссылку двумерный массив элементов `int`, состоящий из двух строк и трех столбцов. В объявлении:

```
const std::array<std::array<int, columns>, rows>&
```

указано, что количество элементов внешнего массива равно `rows` (то есть 2), а также указан их тип:

```
array<int, columns>
```

Таким образом, каждый из элементов внешнего массива — это внутренний массив, состоящий из трех элементов `int` (значение `columns` равно 3). Функция `printArray` не изменяет элементы, поэтому она получает `array` через константную (`const`) ссылку.

Два цикла: внешний и внутренний

Обработка элементов двумерного массива выполняется двумя циклами:

- **внешний цикл (outer loop)** выполняет итерации по строкам;
- **внутренний цикл (inner loop)**, или вложенный цикл, выполняет итерации по столбцам текущей строки.

Эти циклы реализованы двумя инструкциями `for` для диапазона значений. В строках 24 и 26 мы впервые использовали ключевое слово `auto` (C++11). Оно сообщает компилятору, что тип данных переменной надо автоматически определить по типу инициализирующего значения. Во внешнем цикле переменная диапазона `row` инициализируется элементом из параметра `a`. Как видно из объявления массива, этот элемент имеет следующий тип:

```
array<int, columns>
```

Поэтому компилятор определяет, что `row` является ссылкой на трехэлементный массив значений `int` (напомним, что `columns` равно 3). Квалификатор `const` в объявлении `row` запрещает использовать ссылку для изменения элементов и предотвращает копирование каждого элемента в переменную диапазона. Во внутреннем цикле переменная диапазона `element` (тоже объявленная с квалификатором `const`) инициализируется элементом массива, представленным переменной `row`. Компилятор определяет, что `element` является ссылкой на значение `int`, поскольку каждая строка массива содержит три значения `int`. (Во многих интегрированных средах разработки при наведении курсора мыши на переменную, объявленную с ключе-

вым словом auto, отображается ее тип, определенный компилятором.) В строке 27 выводится на экран значение element из соответствующей строки и столбца.

Альтернатива: два цикла for со счетчиком

Циклы for для диапазона значений мы могли бы заменить циклами for со счетчиком:

```
for (size_t row{0}; row < a.size(); ++row) {
   for (size_t column{0}; column < a.at(row).size(); ++column) {
      cout << a.at(row).at(column) << ' '; // Или a[row][column]
   }
   cout << '\n';
}
```

Скобочная инициализация массива массивов

Если вы явно укажете размеры массива при его объявлении, вы можете упростить его список инициализаторов. Например, строку 11 можно было бы записать так:

```
constexpr std::array<std::array<int, columns>, rows> values1{
   {{1, 2, 3}, // Строка 0
    {4, 5, 6}} // Строка 1
};
```

В этом случае, если строка списка инициализаторов содержит меньше элементов, чем количество столбцов, остальные элементы строки инициализируются значением по умолчанию.

Операции с двумерным массивом: установка значений строки

Давайте рассмотрим несколько типичных операций с массивом a, состоящим из трех строк и четырех столбцов, который показан на диаграмме в начале раздела 6.13. Следующий цикл for обнуляет все элементы строки 2:

```
for (size_t column{0}; column < a.at(2).size(); ++column) {
   a.at(2).at(column) = 0; // Или a[2][column]
}
```

Здесь в цикле for изменяется только второй индекс (то есть индекс столбца). Этот код, состоящий из одной инструкции цикла for, эквивалентен четырем инструкциям присваивания:

```
a.at(2).at(0) = 0; // Или a[2][0] = 0;
a.at(2).at(1) = 0; // Или a[2][1] = 0;
a.at(2).at(2) = 0; // Или a[2][2] = 0;
a.at(2).at(3) = 0; // Или a[2][3] = 0;
```

Операции с двумерным массивом: суммирование элементов в циклах for со счетчиком

Следующий код, основанный на внешнем и внутреннем циклах for со счетчиком, суммирует значения элементов массива a:

```
int total{0};
for (size_t row{0}; row < a.size(); ++row) {
   for (size_t column{0}; column < a.at(row).size(); ++column) {
      total += a.at(row).at(column); // a[row][column]
   }
}
```

Здесь все элементы массива суммируются построчно. В первой итерации внешнего цикла индекс строки (`row`) равен `0`, поэтому во внутреннем цикле суммируются элементы строки `0`. Во второй итерации внешнего цикла `row` увеличивается до `1` и во внутреннем цикле суммируются элементы строки `1`. В последней итерации внешнего цикла `row` увеличивается до `2` и во внутреннем цикле суммируются элементы строки `2`.

Операции с двумерным массивом: суммирование элементов в циклах for для диапазона значений

По сравнению с предыдущим примером для суммирования значений элементов массива `a` предпочтительнее следующий код, основанный на внешнем и внутреннем циклах `for` для диапазона значений:

```
int total{0};
for (const auto& row : a) { // Для каждой строки (row) массива a
   for (const auto& column : row) { // Для каждого столбца (column) в строке
      total += column;
   }
}
```

Многомерные массивы mdarray (C++23)

Комитет по стандартам C++ работает над специализированным контейнером многомерных массивов под названием `mdarray`, который может быть реализован в C++23. Здесь вы можете следить за процессом реализации `mdarray` и соответствующего типа `mdspan`:

- `mdarray` — https://isocpp.org/files/papers/D1684R0.html
- `mdspan` — https://wg21.link/P0009

6.14. Основы функционального программирования

Как и другие популярные языки (в том числе Python, Java и C#), язык C++ поддерживает несколько парадигм (концепций, стилей) программирования, которые вы можете комбинировать в вашем коде: процедурное, объектно-ориентированное, обобщенное (шаблонное) и функциональное программирование. Имеющиеся в C++ средства программирования в функциональном стиле помогают писать лаконичный и надежный код, который легко читать, отлаживать и модифицировать. Кроме того, функциональный стиль программирования упрощает распараллеливание программ для ускорения их работы на компьютерах с многоядерными процессорами.

6.14.1. «Что» и «как»

По мере усложнения программы ее код становится очень трудным для понимания, отладки и модификации. Повышается вероятность ошибок. Вам становится все труднее сформулировать, как должен работать код. Программируя в функциональном стиле, вы обычно указываете, что именно вы хотите сделать, а библиотечный код без вашего участия определяет, как это будет сделано. Это предотвращает многие ошибки.

Рассмотрим следующую инструкцию `for` для диапазона значений (из примера на илл. 6.5), суммирующую все целые числа из массива:

```
for (const int& item : integers) {
   total += item;
}
```

Хотя этот процедурный код скрывает детали итерации, нам все равно пришлось указать, как программа должна производить вычисления: «прибавлять каждое значение `item` к значению `total`». Каждый раз, когда вы изменяете какую-либо переменную, вы рискуете ошибиться. Функциональное программирование дает преимущество **неизменяемости (immutability)**: вы можете обойтись без операций, изменяющих значения переменных. При первом знакомстве с этой концепцией вы можете задаться вопросом: «Разве такое возможно?». Читайте дальше.

Свертка в функциональном стиле с помощью accumulate

На илл. 6.11 вы видите пример решения той же задачи сложения всех целых чисел из массива, но уже в функциональном стиле. Вместо инструкции `for` для диапазона значений (из примера на илл. 6.5) мы используем алгоритм (то есть функцию, выполняющую алгоритм) `accumulate` из заголовка `<numeric>` стандартной библиотеки C++. Эта функция знает, как вычислить сумму диапазона значений, сведя их все в единое целое. Это вычисление является примером **свертки (reduction)** – одной из базовых операций в функциональном программировании.

```
 1 // fig06_11.cpp
 2 // Вычисление суммы элементов массива алгоритмом accumulate
 3 #include <array>
 4 #include <fmt/format.h>
 5 #include <iostream>
 6 #include <numeric>
 7
 8 int main() {
 9    constexpr std::array integers{10, 20, 30, 40};
10    std::cout << fmt::format("Total of array elements: {}\n",
11    std::accumulate(std::begin(integers), std::end(integers), 0));
12 }
```

```
Total of array elements: 100
```

Илл. 6.11 | Вычисление суммы элементов массива алгоритмом accumulate

Как и в функции `sort` из раздела 6.12, первые два аргумента функции `accumulate` (строка 11) определяют диапазон элементов для вычисления суммы. В данном случае это все элементы массива `integers` — от начала (`begin`) до конца (`end`). Функция **внутренне (internally)** суммирует обрабатываемые элементы массива, скрывая от вас вычислительный процесс. Третий аргумент — это начальное значение суммы (`0`). У функции `accumulate` есть и другие возможности, о которых вы скоро узнаете.

Функция `accumulate` использует **внутреннюю итерацию (internal iteration)**, которая тоже скрыта от вас. Функция знает, как перебирать диапазон элементов массива и прибавлять значение каждого элемента к текущей сумме. Вы просто указываете, что вы хотите сделать, а библиотека C++ самостоятельно определяет способ выполнения. Это так называемое **декларативное программирование (declarative programming)** — еще одна отличительная черта функционального стиля.

6.14.2. Передача функций в качестве аргументов другим функциям, основы лямбда-выражений

Многие стандартные библиотечные функции C++ принимают в качестве аргументов другие функции, тем самым позволяя вам уточнять, как они должны работать. Функции, принимающие другие функции в качестве аргументов, называются **функциями высшего порядка (higher-order functions)** и часто используются в функциональном программировании. Рассмотрим функцию `accumulate`, которая по умолчанию суммирует значения элементов. Ее можно перегружать, передавая четвертый аргумент: функцию, которая задает алгоритм выполнения свертки. На илл. 6.12 вы видите, как вместо суммы значений перегруженная функция `accumulate` вычисляет их произведение.

```
 1 // fig06_12.cpp
 2 // Вычисление произведения элементов массива алгоритмом accumulate
 3 #include <array>
 4 #include <fmt/format.h>
 5 #include <iostream>
 6 #include <numeric>
 7
 8 int multiply(int x, int y) {
 9    return x * y;
10 }
11
12 int main() {
13    constexpr std::array integers{1, 2, 3, 4, 5};
14
15    std::cout << fmt::format("Product of integers: {}\n", std::accumulate(
16       std::begin(integers), std::end(integers), 1, multiply));
17
18    std::cout << fmt::format("Product of integers with a lambda: {}\n",
19       std::accumulate(std::begin(integers), std::end(integers), 1,
```

Окончание ⇨

```
20          [](const auto& x, const auto& y){return x * y;}));
21 }
```

```
Product of integers: 120
Product of integers with a lambda: 120
```

Илл. 6.12 | Вычисление произведения элементов массива алгоритмом accumulate

Вызов accumulate с именованной функцией в качестве аргумента

В строках 15–16 функция `accumulate` вызывается для массива `integers` (определенного в строке 13). Мы вычисляем произведение, поэтому третий аргумент (начальное значение для свертки) равен `1`, а не `0`, иначе результат был бы равен `0`. Четвертый аргумент — это функция, вызываемая для каждого элемента массива, в данном случае функция `multiply` (определенная в строках 8–10). Она перемножает два числа, полученные в качестве аргументов:

- промежуточный результат;
- значение из массива.

Затем она возвращает значение, которое становится новым промежуточным результатом. Когда функция `accumulate` перебирает значения массива `integers`, она передает функции `multiply` в качестве аргументов промежуточный результат и следующий элемент массива. В этом примере происходит пять внутренних вызовов функции `multiply` из функции `accumulate`:

- Первый вызов передает функции `multiply` начальное значение свертки (число `1`, которое было третьим аргументом функции `accumulate`) и первый элемент массива (`1`). Возвращаемое значение: `1`.
- Второй вызов передает функции `multiply` промежуточный результат (`1`) и второй элемент массива (`2`). Возвращаемое значение: `2`.
- Третий вызов передает функции `multiply` промежуточный результат (`2`) и третий элемент массива (`3`). Возвращаемое значение: `6`.
- Четвертый вызов передает функции `multiply` промежуточный результат (`6`) и четвертый элемент массива (`4`). Возвращаемое значение: `24`.
- Пятый вызов передает функции `multiply` промежуточный результат (`24`) и пятый элемент массива (`5`). Возвращаемое значение: `120`. Это конечный результат свертки, который функция `accumulate` возвращает в точку своего вызова.

Вызов функции accumulate с лямбда-выражением

Иногда вам может понадобиться «одноразовая» функция, которую вы не будете использовать повторно. В таком случае вы можете определить функцию там, где она необходима, используя **лямбда-выражение (lambda expression)**, или, сокращенно, **лямбду**. Лямбда-выражение (C++11) — это, по сути, анонимная функция,

то есть функция без имени. Вызов `accumulate` в строках 19–20 содержит лямбда-выражение, выполняющее ту же задачу, что и функция `multiply`:

```
[](const auto& x, const auto& y){return x * y;}
```

Каждое лямбда-выражение начинается со **списка захвата (capture list)**, который также называют **вводной секцией** или **лямбда-интродуктором (lambda introducer)**, — это пара квадратных скобок с необязательными параметрами внутри (см. главу 14). После списка захвата следует список параметров, разделенных запятыми, и тело функции. В данном случае лямбда получает два параметра, вычисляет их произведение и возвращает результат.

В разделе 6.13 вы видели, что ключевое слово `auto` сообщает компилятору о необходимости автоматически определить тип переменной по ее инициализирующему значению. Указание типа параметра лямбды как `auto` позволяет компилятору определить тип этого параметра на основе контекста, в котором вызывается лямбда. В этом примере функция `accumulate` вызывает лямбду один раз для каждого элемента массива, передавая в качестве аргументов промежуточный результат и значение элемента. Тип начального произведения (`1`) — `int`, и массив тоже содержит элементы `int`, поэтому компилятор определяет типы параметров как `int`. Возможность автоматического определения типа каждого параметра лямбды с помощью ключевого слова `auto` появилась в C++14, такие лямбды называются **обобщенными (generic)**. В нашем примере компилятор автоматически определяет и тип возвращаемого лямбдой значения, основываясь на выражении `x * y`. Здесь `x` и `y` имеют тип `int`, поэтому лямбда возвращает значение `int`.

Мы объявили параметры как константные ссылки, и таким образом:

- поскольку это константы, тело лямбда-выражения не может изменять переменные в вызывающем коде;
- поскольку это ссылки, программа не тратит время на копирование объектов и работает быстрее (это особенно важно при вызовах лямбда-выражений с большими объектами).

Лямбда-выражение из этого примера можно использовать с любым типом данных, поддерживающим оператор *. В главе 14 лямбда-выражения рассматриваются подробно.

6.14.3. Фильтрация, преобразование и свертка с библиотекой ranges (C++20)

Стандартная библиотека C++ уже много лет позволяет программировать в функциональном стиле. В C++20 появилась библиотека `ranges`[1] (заголовок `<ranges>`), упростившая программирование в функциональном стиле. Мы представим вам два ключевых понятия этой библиотеки: диапазоны и представления.

[1] На момент написания книги компилятор clang++ еще не поддерживает представленные здесь функции этой библиотеки.

- **Диапазон (range)** — это набор элементов, которые вы можете поочередно перебирать итерациями. Например, массив — это диапазон[1].
- **Представление (view)** позволяет вам определить операцию, обрабатывающую диапазон. Представления можно комбинировать: вы можете последовательно соединить несколько представлений, чтобы обрабатывать элементы диапазона несколькими операциями.

Наша следующая программа, разделенная на части для удобства обсуждения (илл. 6.13), демонстрирует несколько операций функционального программирования, использующих библиотеку ranges. В главе 14 рассматриваются другие возможности этой библиотеки.

Лямбда showValues, выводящая результаты на экран

В этом примере мы выводим на экран результаты различных операций с диапазонами, используя цикл for для диапазона значений. Вместо многократного повторения кода цикла можно определить функцию, которая получает диапазон и выводит на экран его значения. Мы решили определить обобщенную лямбду (строки 12–20), чтобы показать, что вы можете сохранить лямбду в локальной переменной (showValues; строки 11–21). Затем вы можете вызывать лямбду по имени этой переменной, как в строках 24, 29, 34, 40 и 51.

```
1 // fig06_13.cpp
2 // Функциональное программирование с диапазонами и представлениями (C++20)
3 #include <array>
4 #include <fmt/format.h>
5 #include <iostream>
6 #include <numeric>
7 #include <ranges>
8
9 int main() {
10    // Лямбда для вывода на экран результатов операций с диапазонами
11    auto showValues{
12       [](auto& values, const std::string& message) {
13          std::cout << fmt::format("{}: ", message);
14
15          for (const auto& value : values) {
16             std::cout << fmt::format("{} ", value);
17          }
18
19          std::cout << '\n';
20       }
21    };
22
```

Илл. 6.13 | Функциональное программирование с диапазонами и представлениями (C++20)

[1] Слово «range» в этом контексте для носителей английского языка означает «ряд» или «множество» («ряд значений», «множество чисел»), но в русском языке такой набор элементов обычно называют диапазоном. Авторы книги имеют в виду, что массив — это множество, состоящее из элементов. — *Примеч. ред.*

Генерация диапазона целых чисел с помощью views::iota

В нескольких примерах этой главы мы создавали массив и обрабатывали его элементы. Для этого нам надо было каждый раз начинать с объявления массива с заданным количеством элементов. В некоторых случаях вы можете генерировать значения **по требованию (on demand)** вместо того, чтобы создавать и сохранять их заранее. Операции, генерирующие значения по требованию, используют **ленивые вычисления (lazy evaluation)**, которые уменьшают потребление памяти программой и повышают быстродействие в тех случаях, когда не нужны все значения сразу. В строке 23 мы используем фабрику[1] `views::iota` из библиотеки `ranges` для генерации диапазона последовательных целых чисел от первого аргумента (`1`) до второго аргумента (`11`), но кроме второго аргумента (то есть без числа `11`; это называется полуоткрытым диапазоном). Значения не генерируются, пока не понадобятся программе, например, при вызове лямбды `showValues` (строка 24).

```
23    auto values1{std::views::iota(1, 11)}; // Генерируем числа от 1 до 10
24    showValues(values1, "Generate integers 1-10");
25
```

```
Generate integers 1-10: 1 2 3 4 5 6 7 8 9 10
```

Фильтрация элементов с помощью views::filter

Одна из базовых операций в функциональном программировании — **фильтрация (filtering)** элементов. В результате фильтрации из диапазона выбираются только те элементы, которые соответствуют заданному условию. Как правило, количество элементов после фильтрации меньше, чем в исходном диапазоне. Один из способов реализовать фильтрацию — перебирать элементы в цикле, проверяя каждый из них вложенной инструкцией `if` на соответствие заданному условию. По итогам проверки элемента с ним можно что-то сделать, например добавить в контейнер. Этот способ требует явного определения инструкции цикла и изменяемых переменных, что может приводить к ошибкам, как мы отмечали в разделе 6.6.

Работая с диапазонами и представлениями, мы можем реализовать фильтрацию с помощью адаптера[2] `views::filter`, чтобы сосредоточиться на том, что мы хотим получить. В данном случае мы хотим получить только четные числа от `1` до `10`. Инициализатор `values2` в строках 27–28 использует оператор `|` для соединения нескольких операций. Первая операция (`values1` из строки 23) генерирует числа от `1` до `10`, а вторая фильтрует результат первой. Вместе эти операции образуют **конвейер (pipeline)**. Каждый конвейер начинается с диапазона, который является источником данных (в нашем случае это `values1`), за которым следует произвольное количество операций, разделенных символом `|`.

---

[1] Этот термин объясняется в разделе 14.7.2. — *Примеч. ред.*

[2] Этот термин объясняется в разделе 14.7.1. — *Примеч. ред.*

```
26    // Фильтруем значения values1, сохраняя только четные числа
27    auto values2{values1 |
28       std::views::filter([](const auto& x) {return x % 2 == 0;})};
29    showValues(values2, "Filtering even integers");
30
```

```
Filtering even integers: 2 4 6 8 10
```

Аргумент адаптера `views::filter` должен быть функцией, которая получает одно значение для обработки и возвращает результат типа `bool`, указывающий, следует ли сохранять это значение. Мы передали в качестве аргумента лямбду, которая возвращает `true`, если ее аргумент делится без остатка на 2.

Как только будут выполнены строки 27–28, `values2` становится **ленивым конвейером (lazy pipeline)**, который генерирует целые числа от `1` до `10` и фильтрует их, сохраняя только четные числа. Конвейер кратко определяет то, что мы хотим сделать (а не как это сделать):

- Фабрика `views::iota` знает, как генерировать целые числа.
- Адаптер `views::filter` знает, как использовать свою функцию-аргумент, чтобы определять, когда надо сохранять элементы, полученные из предыдущей части конвейера.

Этот конвейер называется ленивым, потому что он не производит никаких результатов, пока нет обращения к `values2`.

Как только лямбда `showValues` начинает перебирать элементы `values2`, фабрика `views::iota` генерирует первое значение, а затем адаптер `views::filter` вызывает функцию-аргумент, чтобы определить, надо ли сохранить это значение. Если эта функция возвращает `true`, инструкция `for` для диапазона значений (входящая в `showValues`) получает значение из конвейера и выводит его на экран. В противном случае шаги обработки повторяются со следующим значением, которое генерирует фабрика `views::iota`.

Маппинг элементов с помощью views::transform

Еще одна базовая операция в функциональном программировании — это **маппинг (mapping)** диапазона, то есть преобразование его элементов по единому алгоритму в новые значения, иногда другого типа. В результате маппинга количество элементов в диапазоне не изменяется. В библиотеке `ranges` для такого преобразования элементов есть адаптер `views::transform`. Конвейер в строках 32–33 добавляет к конвейеру из строк 27–28 еще одну операцию: лямбда-выражение, переданное адаптеру `views::transform` в строке 33, возводит во вторую степень все отфильтрованные числа из `values2`.

```
31 // Возводим в квадрат каждое число из values2
32    auto values3{
```

```
33       values2 | std::views::transform([](const auto& x) {return x * x;})};
34    showValues(values3, "Mapping even integers to squares");
35
```

```
Mapping even integers to squares: 4 16 36 64 100
```

Аргумент адаптера `views::transform` — это функция, которая получает значение для обработки и возвращает новое значение — возможно, другого типа. Когда `showValues` начинает перебирать результаты конвейера `values3`:

1. Фабрика `views::iota` создает значение.
2. Адаптер `views::filter` определяет, является ли это значение четным. Если да, значение передается на шаг 3. Если нет, обрабатывается следующее значение, которое генерирует фабрика `views::iota` на шаге 1.
3. Адаптер `views::transform` вычисляет квадрат четного целого числа (лямбда в строке 33), затем инструкция `for` для диапазона значений (входящая в `showValues`) отображает результат и обработка продолжается со следующим значением, которое генерирует фабрика `views::iota` на шаге 1.

Объединение в конвейер операций фильтрации и преобразования

Конвейер может содержать произвольное количество операций, разделенных операторами |. Конвейер в строках 37–39 объединяет все предыдущие операции, а в строке 40 выводятся результаты.

```
36 // Объединяем фильтрацию и преобразование, получаем квадраты четных чисел
37    auto values4{
38       values1 | std::views::filter([](const auto& x) {return x % 2 == 0;})
39               | std::views::transform([](const auto& x) {return x * x;})};
40    showValues(values4, "Squares of even integers");
41
```

```
Squares of even integers: 4 16 36 64 100
```

Свертка конвейера диапазона функцией accumulate

Функции стандартной библиотеки C++, такие как `accumulate`, тоже работают с ленивыми конвейерами диапазонов. В строке 44 выполняется свертка, суммирующая квадраты четных целых чисел, полученные конвейером в строках 37–39.

```
42    // Суммируем квадраты четных чисел
43    std::cout << fmt::format("Sum squares of even integers 2-10: {}\n",
44       std::accumulate(std::begin(values4), std::end(values4), 0));
45
```

```
Sum squares of even integers 2-10: 220
```

Фильтрация и преобразование элементов контейнера

Различные контейнеры C++, в том числе массивы и векторы (раздел 6.15), могут быть источниками данных в конвейерах диапазонов. В строке 47 создается массив, содержащий числа от **1** до **10**. Затем он используется в конвейере, вычисляющем квадраты четных целых чисел из этого массива.

```
46    // Обработка элементов контейнера
47    constexpr std::array numbers{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
48    auto values5{
49       numbers | std::views::filter([](const auto& x) {return x % 2 == 0;})
50               | std::views::transform([](const auto& x) {return x * x;})};
51    showValues(values5, "Squares of even integers in array numbers");
52 }
```

```
Squares of even integers in array numbers: 4 16 36 64 100
```

6.15. Готовые объекты: шаблон класса vector стандартной библиотеки C++

Теперь, продолжая презентацию готовых объектов, мы расскажем об объектах шаблона класса `vector`[1] стандартной библиотеки C++ (заголовок `<vector>`). Векторы[2] схожи с массивами, но размеры векторов можно динамически менять. В конце этого раздела мы покажем, как проверять границы векторов (такая проверка возможна и для массивов). Мы рассмотрим имеющийся в C++ механизм обработки исключений при попытке доступа к несуществующему элементу вне границ вектора. Для этого мы включили в программу (илл. 6.14) заголовок `<stdexcept>` (строка 5). В строках 7–8 объявлены прототипы функций `outputVector` (строки 93–99) и `inputVector` (строки 102–106), которые служат для вывода и ввода значений элементов вектора соответственно.

```
1 // fig06_14.cpp
2 // Демонстрация шаблона класса vector стандартной библиотеки C++
3 #include <iostream>
4 #include <vector>
5 #include <stdexcept>
6
7 void outputVector(const std::vector<int>& items); // Функция для вывода
8 void inputVector(std::vector<int>& items); // Функция для ввода
9
```

Илл. 6.14 | Демонстрация шаблона класса vector стандартной библиотеки C++

[1] Другие возможности векторов обсуждаются в главе 13.

[2] Векторы языка C++ иногда называют «динамическими массивами класса vector» (сокращенно — «динамическими массивами»), если надо подчеркнуть, что речь идет не о векторах в математическом или ином значении этого слова. — *Примеч. ред.*

Создание объектов класса vector

В строках 11–12 создаются два вектора с элементами типа `int`: вектор `integers1`, содержащий `7` элементов, и вектор `integers2`, содержащий `10` элементов. По умолчанию все элементы каждого созданного вектора инициализируются значением `0`. Как и массивы, векторы могут хранить большинство типов данных — вы можете просто заменить `int` в `std::vector<int>` на нужный вам тип.

```
10 int main() {
11    std::vector<int> integers1(7); // Вектор с семью элементами
12    std::vector<int> integers2(10); // Вектор с десятью элементами
13
```

Обратите внимание, что аргументы, обозначающие размеры векторов (`7` и `10`), передаются конструкторам векторов в круглых скобках (а не в фигурных). Если при создании вектора поместить значение в фигурные скобки, компилятор воспримет это как список инициализаторов с одним элементом, а не как размер вектора. Например, следующее объявление создает вектор, содержащий один элемент со значением 7, а не вектор из семи элементов:

```
std::vector<int> integers1{7};
```

Если содержимое вектора известно на этапе компиляции, вы можете определить вектор с помощью списка инициализаторов и дедуктивного определения аргументов шаблона класса (CTAD), как мы это делали с массивами. Например, следующая инструкция определяет вектор с четырьмя элементами типа `double`:

```
std::vector integers1{1.1, 2.2, 3.3, 4.4};
```

Функции size и outputVector

В строке 15 вызывается функция `size` класса `vector`, возвращающая количество элементов вектора `integers1`[1]. В строке 17 вектор `integers1` передается пользовательской функции `outputVector` (строки 93–99), которая выводит на экран элементы вектора, используя инструкцию `for` для диапазона значений. В строках 20 и 22 выполняются аналогичные действия для вектора `integers2`.

```
14    // Выводим размер и содержимое вектора integers1
15    std::cout << "Size of vector integers1 is " << integers1.size()
16       << "\nvector after initialization:";
17    outputVector(integers1);
18
19    // Выводим размер и содержимое вектора integers2
```

Окончание ⇨

[1] Вы также можете использовать глобальную функцию C++17 std::size, например std::size(integers1).

```
20    std::cout << "\nSize of vector integers2 is " << integers2.size()
21       << "\nvector after initialization:";
22    outputVector(integers2);
23
```

```
Size of vector integers1 is 7
vector after initialization: 0 0 0 0 0 0 0

Size of vector integers2 is 10
vector after initialization: 0 0 0 0 0 0 0 0 0 0
```

Функция inputVector

В строках 26–27 векторы `integers1` и `integers2` передаются функции `inputVector` (строки 102–106), которая получает значения элементов, вводимые пользователем.

```
24    // Считываем значения для integers1 и integers2, выводим их на экран
25    std::cout << "\nEnter 17 integers:\n";
26    inputVector(integers1);
27    inputVector(integers2);
28
29    std::cout << "\nAfter input, the vectors contain:\n"
30       << "integers1:";
31    outputVector(integers1);
32    std::cout << "integers2:";
33    outputVector(integers2);
34
```

```
Enter 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

After input, the vectors contain:
integers1: 1 2 3 4 5 6 7
integers2: 8 9 10 11 12 13 14 15 16 17
```

Сравнение векторов оператором !=

В главе 5 мы рассказали о перегрузке функций. Аналогичная концепция — перегрузка операторов, позволяющая вам определять, как стандартные операторы C++ должны работать с пользовательскими типами данных. В стандартной библиотеке C++ уже есть определения перегруженных операторов == и != для работы с массивами и векторами. Поэтому вы можете сравнивать этими операторами два массива или два вектора на равенство или неравенство соответственно. В строке 38 два вектора сравниваются оператором !=. Он возвращает значение `true`, если векторы отличаются (если у них разный размер или какие-то их элементы с одинаковыми индексами не совпадают). В противном случае != возвращает `false`.

```
35    // Сравниваем векторы оператором неравенства (!=)
36    std::cout << "\nEvaluating: integers1 != integers2\n";
37
```

```
38    if (integers1 != integers2) {
39       std::cout << "integers1 and integers2 are not equal\n";
40    }
41
```

```
Evaluating: integers1 != integers2
integers1 and integers2 are not equal
```

Инициализация вектора содержимым другого вектора

Вы можете инициализировать новый вектор, скопировав в него содержимое существующего вектора. В строке 44 создается вектор `integers3` и инициализируется копией вектора `integers1`. Здесь мы использовали CTAD для автоматического определения типа `integers3` по типу `integers1`. Для выполнения операции копирования в строке 44 вызывается копирующий конструктор шаблона класса `vector`. Мы расскажем о копирующих конструкторах в главе 11. В строках 46–48 выводятся на экран размеры и содержимое `integers3`, что подтверждает правильную инициализацию.

```
42    // Создаем вектор integers3, используя integers1 как инициализатор;
43    // выводим на экран размер и содержимое вектора integers3
44    std::vector integers3{integers1}; // Копирующий конструктор
45
46    std::cout << "\nSize of vector integers3 is " << integers3.size()
47       << "\nvector after initialization: ";
48    outputVector(integers3);
49
```

```
Size of vector integers3 is 7
vector after initialization: 1 2 3 4 5 6 7
```

Присваивание векторов и сравнение векторов

В строке 52 вектор `integers2` присваивается вектору `integers1`, чтобы показать, что оператор присваивания (`=`) перегружен для работы с векторами. В строках 54–57 выводятся на экран элементы обоих векторов, и вы видите, что они одинаковы. Затем в строке 62 векторы `integers1` и `integers2` сравниваются оператором равенства (`==`), чтобы определить, идентично ли их содержимое после присваивания.

```
50    // Используем перегруженный оператор присваивания (=)
51    std::cout << "\nAssigning integers2 to integers1:\n";
52    integers1 = integers2; // Присваиваем вектор integers2 вектору integers1
53
54    std::cout << "integers1: ";
55    outputVector(integers1);
56    std::cout << "integers2: ";
57    outputVector(integers2);
58
59    // Сравниваем векторы оператором равенства (==)
```

Окончание ⇨

```
60    std::cout << "\nEvaluating: integers1 == integers2\n";
61
62    if (integers1 == integers2) {
63       std::cout << "integers1 and integers2 are equal\n";
64    }
65
```

```
Assigning integers2 to integers1:
integers1: 8 9 10 11 12 13 14 15 16 17
integers2: 8 9 10 11 12 13 14 15 16 17

Evaluating: integers1 == integers2
integers1 and integers2 are equal
```

Функция at для доступа к элементам вектора

В строке 67 вызывается функция `at` класса `vector` для доступа к заданному элементу вектора и для получения его значения. В строке 71 эта функция вызывается для изменения значения элемента. Если функция `at` вызвана для элемента с корректным (существующим в векторе) индексом, она возвращает:

- ссылку на элемент, которую можно использовать для изменения его значения; или
- константную ссылку на элемент, которую нельзя использовать для изменения его значения (но можно использовать для получения его значения).

Функция `at` возвращает константную ссылку, если вызывается для вектора, объявленного с квалификатором `const`, или через константную ссылку.

```
66    // Используем значение с индексом 5 как rvalue
67    std::cout << "\nintegers1.at(5) is " << integers1.at(5);
68
69    // Используем integers1.at(5) как lvalue
70    std::cout << "\n\nAssigning 1000 to integers1.at(5)\n";
71    integers1.at(5) = 1000;
72    std::cout << "integers1: ";
73    outputVector(integers1);
74
```

```
integers1.at(5) is 13

Assigning 1000 to integers1.at(5)
integers1: 8 9 10 11 12 1000 14 15 16 17
```

Как и массивы, векторы работают с оператором `[]`. Согласно стандарту C++, при доступе к элементам через квадратные скобки компилятор не обязан выполнять проверку границ. Поэтому вы должны убедиться, что ни одна инструкция с оператором `[]` не может случайно обратиться к несуществующему элементу вне границ вектора.

Обработка исключений: обращение к элементу вне границ вектора

В строке 78 мы пробуем вывести на экран значение элемента `integers1.at(15)`, которого нет в векторе. Функция `at` при проверке границ обнаруживает недопустимый индекс и **выбрасывает**[1] **(throws)** исключение, указывающее на ошибку во время выполнения. Слово «исключение» подразумевает, что это редкая, исключительная ситуация. **Обработка исключений (exception handling)** позволяет создавать **отказоустойчивые программы (fault-tolerant programs)**, то есть программы, способные корректно обрабатывать (или перехватывать) исключения. В некоторых случаях обработка исключения позволяет программе продолжать выполнение, как будто никакой ошибки не было. Например, наша программа будет нормально работать даже после того, как мы обратились к элементу с недопустимым индексом. Но более серьезная ошибка может прервать нормальное выполнение программы, не давая ей возможности освободить перед завершением все ресурсы, которые она использовала (например, закрыть открытый файл или подключение к базе данных). Здесь мы кратко обсудим обработку исключений. В главе 9 вы узнаете, как выбрасывать исключения из пользовательских функций, а обработку исключений мы подробно рассмотрим в главе 12.

```
75    // Пробуем обратиться к элементу с несуществующим индексом
76    try {
77       std::cout << "\nAttempt to display integers1.at(15)\n";
78       std::cout << integers1.at(15) << '\n'; // Ошибка: индекс вне границ
79    }
80    catch (const std::out_of_range& ex) {
81       std::cerr << "An exception occurred: " << ex.what() << '\n';
82    }
83
```

```
Attempt to display integers1.at(15)
An exception occurred: invalid vector subscript
```

Инструкция try

По умолчанию исключение приводит к аварийному завершению написанной на C++ программы. Чтобы обработать исключение и по возможности продолжить выполнение программы, надо поместить код, который может вызвать исключение, в инструкцию `try` (строки 76–82). Блок `try` (строки 76–79) содержит код, который может вызвать исключение, а блок `catch` (строки 80–82) содержит код, обрабатывающий это исключение, если оно случится. Как вы увидите в главе 12, за одним блоком `try` могут следовать много блоков `catch`, обрабатывающих разные типы предполагаемых исключений. Если код в блоке `try` выполняется успешно,

[1] Программисты говорят, что функция «выбрасывает», «бросает», «выдает», «создает» или «генерирует» исключение, — в таком контексте эти слова являются синонимами. — *Примеч. ред.*

инструкции в строках 80–82 игнорируются. Фигурные скобки, разделяющие тела блоков `try` и `catch`, обязательны.

Выполнение блока catch

Когда программа вызывает функцию `at` с аргументом `15` (строка 78), эта функция пытается получить доступ к элементу с индексом `15`, который не соответствует границам вектора: `integers1` содержит только `10` элементов. Поскольку проверка границ происходит во время выполнения программы, функция `at` генерирует исключение (создает объект исключения). Конкретнее, в строке 78 выдается исключение `out_of_range` (заголовок `<stdexcept>`), чтобы сообщить программе о возникшей проблеме. В этот момент блок `try` немедленно прерывается, то есть все последующие инструкции в нем пропускаются и начинается выполнение блока `catch`. Если вы объявили какие-либо переменные в блоке `try`, они теперь окажутся вне области видимости и будут недоступны в блоке `catch`.

Блок `catch` должен объявлять свой параметр исключения (`ex`) как константную ссылку; мы подробнее поговорим об этом в главе 12. Блок `catch` может обрабатывать исключения указанного типа. Внутри блока вы можете использовать идентификатор этого параметра для взаимодействия с перехваченным объектом исключения.

Функция what объекта исключения

Когда в строках 80–82 перехватывается исключение, программа выводит на экран сообщение (которое зависит от компилятора), указывающее на возникшую проблему. В строке 81 вызывается функция `what` объекта исключения, предоставляющая строку для сообщения об ошибке. В нашем примере сразу после вывода сообщения исключение считается обработанным и программа продолжает выполнение со следующей инструкции после закрывающей фигурной скобки блока `catch`. Далее выполняются строки 85–89.

Изменение размера вектора

Одно из важнейших различий между векторами и массивами заключается в том, что векторы могут динамически увеличиваться и уменьшаться по мере изменения количества элементов. Чтобы продемонстрировать это, в строке 85 выводится размер вектора `integers3`, в строке 86 вызывается функция `push_back`, добавляющая в конец вектора новый элемент со значением `1000`, а в строке 87 выводится новый размер `integers3`. Затем в строке 89 выводится на экран обновленное содержимое вектора.

```
84    // Изменение размера вектора
85    std::cout << "\nCurrent integers3 size is: " << integers3.size();
86    integers3.push_back(1000); // Добавляем 1000 в конец вектора
87    std::cout << "\nNew integers3 size is: " << integers3.size()
88       << "\nintegers3 now contains: ";
89    outputVector(integers3);
90 }
91
```

```
Current integers3 size is: 7
New integers3 size is: 8
integers3 now contains: 1 2 3 4 5 6 7 1000
```

Функции outputVector и inputVector

Функция `outputVector` использует цикл `for` для диапазона значений, чтобы получать значение каждого элемента вектора для вывода на экран. Как и в случае с массивом, то же самое можно было бы сделать и в цикле со счетчиком, но мы рекомендуем обрабатывать векторы в цикле `for` для диапазона значений. Аналогично функция `inputVector` содержит цикл `for` с переменной диапазона типа `int&`, которая используется для сохранения введенного значения в соответствующем элементе вектора.

```
92 // Выводим содержимое вектора
93 void outputVector(const std::vector<int>& items) {
94    for (const int& item : items) {
95       std::cout << item << " ";
96    }
97
98    std::cout << '\n';
99 }
100
101 // Записываем значения в вектор
102 void inputVector(std::vector<int>& items) {
103    for (int& item : items) {
104       std::cin >> item;
105    }
106 }
```

Простой линейный код

Изучая наши примеры кода, вы видели много циклов `for`. Возможно, вы заметили важную особенность разделов, посвященных готовым объектам: те фрагменты кода, в которых создаются и используются объекты, — это в основном линейные последовательности инструкций с небольшим количеством управляющих операторов. При работе с объектами код часто «сглаживается», то есть становится простой последовательностью вызовов функций.

6.16. Итоги

В этой главе вы познакомились со структурами данных, изучая шаблоны классов `array` и `vector` стандартной библиотеки C++, которые предназначены для хранения и обработки данных в списках и таблицах. Мы показали, как объявлять массивы, инициализировать их и ссылаться на отдельные элементы. Мы передавали массивы функциям по ссылке и использовали квалификатор `const`, чтобы предотвратить изменение элементов вызываемыми функциями. Вы узнали, как

использовать инструкцию `for` для диапазона значений (C++11) для обработки элементов массива. Мы также продемонстрировали функции `sort` и `binary_search` стандартной библиотеки C++, предназначенные для сортировки массива и для поиска элемента в отсортированном массиве соответственно.

Вы научились объявлять и обрабатывать двумерные массивы. Мы использовали вложенные циклы `for` для диапазона значений (а также вложенные циклы `for` со счетчиком), чтобы перебирать все строки и столбцы двумерного массива. Мы показали, как компилятор автоматически определяет по значению инициализатора тип переменной, объявленной с ключевым словом `auto`. Мы представили вам примеры функционального программирования, в том числе с диапазонами и представлениями библиотеки `ranges` (C++20) и ленивыми конвейерами.

В разделе, посвященном готовым объектам, мы продемонстрировали возможности шаблона класса `vector` стандартной библиотеки C++. Мы обсудили средства проверки границ массивов и векторов при доступе к их элементам, а также основные принципы обработки исключений. В последующих главах мы продолжим рассмотрение структур данных.

В главе 7 мы расскажем об указателях, которые относятся к самым мощным средствам языка C++. Указатели отслеживают, где данные хранятся в памяти, открывая перед программистом интересные возможности для обработки этих данных. Как вы увидите, в C++ сохранилась поддержка традиционных массивов (сильно отличающихся от `array`), тесно связанных с указателями. Но в современном коде вместо них лучше использовать безопасные массивы на основе шаблона класса `array`.



Указатели в современном C++ (фактор риска)

В этой главе вы:

- познакомитесь с указателями, научитесь их объявлять и инициализировать;
- познакомитесь с операторами взятия адреса (&) и косвенного обращения (*);
- сравните возможности указателей и ссылок;
- научитесь передавать аргументы функции по указателю;
- узнаете о традиционных массивах и о строках на основе указателей, которые встречаются в устаревшем коде;
- научитесь использовать квалификатор `const` с указателями и данными, на которые они указывают;
- научитесь использовать оператор `sizeof` для определения количества байтов, хранящих значение определенного типа;
- научитесь понимать выражения и арифметические операции с указателями, которые встречаются в устаревшем коде;
- познакомитесь с ключевым словом `nullptr` (C++11), обозначающим нулевой указатель (указывающий на «ничто»);
- научитесь использовать с традиционными массивами функции `begin` и `end` стандартной библиотеки (C++11);
- узнаете, почему в C++ Core Guidelines рекомендуется избегать указателей и традиционных массивов;
- научитесь преобразовывать традиционные массивы и списки инициализаторов в массивы `std::array` с помощью функции `to_array` (C++20);
- продолжите изучение готовых объектов, безопасно обращаясь к массивам и векторам через представления, созданные с помощью шаблона класса `span` (C++20).

7.1. Введение

В этой главе рассматриваются **указатели (pointers)** и тесно связанные с ними элементы языка C++:

- традиционные **массивы на основе указателей (pointer-based arrays)**, также называемые **встроенными массивами (built-in arrays)** или **массивами в стиле C (C-style arrays)**;
- **строки на основе указателей (pointer-based strings)**, также называемые **строками в стиле C (C-strings)**. Все эти элементы C++ унаследованы от языка C.

Неактуальность указателей в современных программах на C++

Указатели — это мощные средства программирования, но с ними трудно работать, что часто приводит к ошибкам. В современном C++ (начиная с C++11) есть другие средства, устраняющие необходимость в большинстве указателей. В новых проектах по разработке программного обеспечения, как правило, надо отдавать предпочтение:

- ссылкам или объектам «умных указателей» (раздел 11.5) вместо указателей;
- объектам `std::array` и `std::vector` (глава 6) вместо традиционных массивов;
- объектам классов `std::string` (главы 2 и 8) и `std::string_view` (раздел 8.11) вместо строк в стиле C.

Иногда указатели все еще нужны

При изучении, доработке и использовании устаревшего кода вы часто будете встречать указатели, традиционные массивы и строки в стиле C. Указатели нужны для того, чтобы:

- создавать и использовать динамические структуры данных (например, связные списки, очереди, стеки и деревья), которые могут увеличиваться и уменьшаться во время выполнения программы (впрочем, программисты, как правило, предпочитают динамические контейнеры стандартной библиотеки C++: уже известные вам векторы, а также контейнеры, о которых вы узнаете в главе 13);
- обрабатывать аргументы командной строки, которые программа получает в виде традиционного массива строк в стиле C;
- передавать аргументы функциям в тех случаях, когда надо предусмотреть возможность передать `nullptr`[1] (указатель на «ничто», см. раздел 7.2.2); для сравнения, по ссылкам можно передавать только существующие объекты[2].

---

[1] C++ Core Guidelines, «F.60: Prefer T* over T& When "No Argument" Is a Valid Option». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-ptr-ref.

[2] В современном C++ можно представить опциональный объект (такой, который может существовать или не существовать) с помощью std::optional (C++17). https://en.cppreference.com/w/cpp/utility/optional.

Рекомендации C++ Core Guidelines, связанные с указателями

Мы должны подчеркнуть, что в C++ Core Guidelines есть рекомендации избегать указателей, массивов на основе указателей и строк на основе указателей — это повышает надежность и безопасность кода. В частности, этот документ неоднократно рекомендует передавать функциям аргументы не по указателям, а по ссылкам[1].

Альтернативы указателям в C++20

Для тех случаев, когда по-прежнему нужны массивы в стиле C (например, для обработки аргументов командной строки), в C++20 добавлены средства, которые сделают ваши программы более безопасными и надежными:

- функция `to_array` преобразует традиционные массивы в современные массивы `std::array`, имеющие целый ряд преимуществ, о которых мы рассказали в главе 6;
- класс `span` — более безопасный способ передачи традиционных массивов функциям. Он является итерируемым (перебираемым поэлементно), поэтому при обработке элементов вы можете использовать его в инструкции `for` для диапазона значений, не рискуя обратиться к несуществующему элементу за границами массива. Помимо этого, вы можете использовать `span` с имеющимися в стандартной библиотеке алгоритмами обработки контейнеров, например `accumulate` или `sort`. В разделе этой главы, посвященном готовым объектам (раздел 7.10), вы увидите, что `span` также работает с современными массивами `std::array` и векторами `std::vector`.

Главный вывод, который вам надо сделать после прочтения этой главы: старайтесь не включать в ваш код указатели, традиционные массивы и строки в стиле C. Но если вам все-таки придется работать с ними, вспомните о тех ценных возможностях, которые дают `to_array` и `span`.

Другие концепции, представленные в этой главе

Мы начнем с объявления и инициализации указателей, а затем продемонстрируем необходимые для работы с ними операторы `&` и `*`. В главе 5 рассматривалась передача аргументов по ссылке. В этой главе мы рассмотрим передачу аргументов по указателю, которая похожа на передачу по ссылке. После этого мы перейдем к традиционным массивам и покажем их тесную связь с указателями.

Мы рассмотрим все варианты применения квалификатора `const` к указателям и к тем данным, на которые они указывают. Мы представим вам оператор `sizeof` и покажем, как узнать, сколько байтов занимают в памяти вашего компьютера элементы данных основных типов и указатели. Затем мы приведем примеры выражений и арифметических операций с указателями.

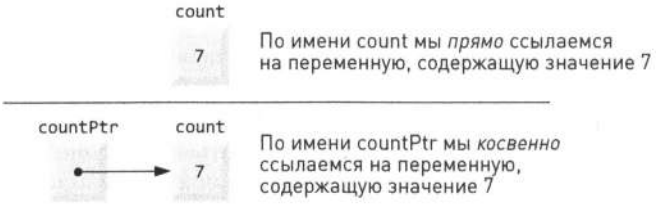
---

[1] C++ Core Guidelines. «F: Functions». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-functions.

В относительно старых программах часто встречаются строки в стиле C. Мы кратко рассмотрим их в этой главе. Вы узнаете, как обрабатывать аргументы командной строки: это простая задача, для выполнения которой в современном C++ по-прежнему нужны строки и массивы в стиле C.

7.2. Объявление и инициализация указателей

Указатель (pointer) — это переменная, значением которой является адрес памяти компьютера. В обычной переменной хранится определенное значение (например, целое число `7`). А в указателе хранится тот адрес, по которому эта переменная расположена в памяти. Можно сказать, что по имени переменной мы прямо ссылаемся на значение 7, а по имени указателя — косвенно, как показано на диаграмме:



Когда мы ссылаемся на значение не прямо, а через указатель, это называется **косвенным обращением (indirection)**.

7.2.1. Объявление указателя

В следующей инструкции мы объявляем переменную `countPtr` типа `int*` (она является указателем на значение типа `int`). Это читается справа налево: «`countPtr` — указатель на `int`»:

```
int* countPtr{nullptr};
```

Символ `*` здесь не является оператором; он означает, что переменная `countPtr` справа от него является указателем. Мы взяли за правило добавлять префикс `Ptr` (сокращение от слова «pointer») к именам всех переменных-указателей. Такое имя позволяет понять, что перед нами указатель и с ним надо работать соответствующим образом.

7.2.2. Инициализация указателя

Каждый указатель надо инициализировать или конкретным адресом памяти, или значением `nullptr` (C++11). Указатель со значением `nullptr` указывает на «ничто» и называется **нулевым указателем (null pointer)**. С этого момента, когда мы говорим о нулевом указателе, мы имеем в виду указатель со значением `nullptr`. Обязательно инициализируйте все указатели, иначе они будут указывать на неизвестные или неинициализированные области памяти.

7.2.3. Нулевые указатели до C++11

В устаревших версиях C++ нулевые указатели имели значение `0` или `NULL`. Значение `NULL` определено в нескольких заголовках стандартной библиотеки для обозначения константы `0`, поэтому инициализация указателя значением `NULL` эквивалентна его инициализации значением `0`. Согласно C++ Core Guidelines, теперь для инициализации указателей нужно использовать `nullptr` вместо `0` или `NULL`[1].

7.3. Операторы для работы с указателями

Оператор `&` создает значение указателя, а оператор `*` выполняет разыменование указателя. В следующих разделах мы объясним, что это значит, и покажем, как использовать эти операторы.

7.3.1. Оператор взятия адреса (&)

Оператор взятия адреса (`&`) — это унарный оператор, который получает адрес памяти своего операнда. Предположим, мы сделали два объявления:

```
int y{5}; // Объявляем переменную y
int* yPtr{nullptr}; // Объявляем указатель yPtr
```

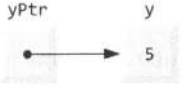
После этого следующая инструкция присвоит указателю `yPtr` адрес переменной `y`:

```
yPtr = &y; // Присваиваем указателю yPtr адрес переменной y в памяти
```

Иными словами, переменная `yPtr` теперь указывает на `y`. Указатель `yPtr` косвенно ссылается на значение переменной `y`.

Оператор `&` в предыдущей инструкции не является объявлением ссылочной переменной (как вы уже знаете, когда объявляется ссылочная переменная, символ `&` добавляется к имени типа и является частью типа). В выражении, подобном `&y`, символ `&` является оператором взятия адреса.

На диаграмме показано состояние двух ячеек памяти после выполнения предыдущей инструкции:



Сформированная указателем связь обозначена стрелкой, которая ведет от ячейки памяти с указателем `yPtr` к ячейке памяти с переменной `y`.

[1] C++ Core Guidelines, «ES.47: Use nullptr Rather Than 0 or NULL». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-nullptr.

На следующей диаграмме показан другой пример состояния двух ячеек памяти. Переменная y типа `int` находится в памяти по адресу `600000`, а указатель `yPtr` — по адресу `500000`.



Операндом оператора **&** должно быть значение *lvalue*. Иными словами, оператор взятия адреса нельзя применять к литералам или к выражениям, которые приводят к временным значениям (например, к результатам вычислений).

7.3.2. Оператор косвенного обращения (*)

Результатом применения унарного оператора * к указателю является значение *lvalue*, представляющее объект, на который указывает указатель. Этот оператор называется **оператором косвенного обращения (indirection operator)** или **оператором разыменования (dereferencing operator)**. Если `yPtr` указывает на переменную y, а переменная y содержит `5` (как на предыдущих диаграммах), то следующая инструкция выведет на экран число `5`:

```
std::cout << *yPtr << '\n';
```

То есть она работает так же, как уже знакомая вам инструкция:

```
std::cout << y << '\n';
```

Такое использование оператора косвенного обращения называется **разыменованием указателя (dereferencing a pointer)**. Разыменованный указатель можно использовать в качестве значения *lvalue* в инструкциях присваивания. Следующая инструкция присваивает значение `9` переменной y:

```
*yPtr = 9;
```

Здесь `*yPtr` — это значение *lvalue*, ссылающееся на переменную y. Аналогичным образом с помощью оператора косвенного обращения можно получить значение из входного потока:

```
cin >> *yPtr;
```

Эта инструкция поместит введенное значение в переменную y.

Неопределенное поведение

Разыменование неинициализированного указателя или указателя, который больше не указывает на объект, приводит к неопределенному поведению программы, которое может проявиться как фатальная ошибка во время выполнения. Неопределенное поведение также может исказить важные данные, то есть программа продолжит выполняться до обычного завершения, но с неправильными результатами. Злоумышленники могут пользоваться этой уязвимостью в программе, чтобы полу-

чать доступ к чужим данным, подменять их и заражать компьютеры вредоносным кодом[1,2,3]. Использование результата разыменования нулевого указателя (`nullptr`), то есть указателя на «ничто», приводит к неопределенному поведению с теми же последствиями. Если вам необходимо работать с указателями, проверяйте перед каждым разыменованием, что указатель не является `nullptr`.

7.3.3. Использование операторов взятия адреса (&) и косвенного обращения (*)

На илл. 7.1 показан пример с операторами `&` и `*`, которые имеют третий по старшинству приоритет. Полная таблица приоритетов операторов приведена в приложении А. В этом примере адреса памяти выводятся оператором `<<` в виде шестнадцатеричных целых чисел (чисел с основанием 16). Выходные данные показывают, что адрес переменной `a` (строка 9) и значение указателя `aPtr` (строка 10) идентичны. Это подтверждает, что адрес переменной `a` действительно был присвоен указателю `aPtr` (строка 7). Выходные данные в строках 11–12 подтверждают идентичность значений `*aPtr` и `a`. Отображаемые адреса памяти зависят от компилятора и платформы. Обычно они меняются при каждом выполнении программы, поэтому вы, скорее всего, увидите другие адреса.

```
1 // fig07_01.cpp
2 // Пример работы с операторами & и *
3 #include <iostream>
4
5 int main() {
6    constexpr int a{7}; // Инициализируем a значением 7
7    const int* aPtr{&a}; // Инициализируем aPtr адресом переменной a
8
9    cout << "The address of a is " << &a
10      << "\nThe value of aPtr is " << aPtr;
11   cout << "\n\nThe value of a is " << a
12      << "\nThe value of *aPtr is " << *aPtr << '\n';
13 }
```

```
The address of a is 000000D649EFFD00
The value of aPtr is 000000D649EFFD00

The value of a is 7
The value of *aPtr is 7
```

Илл. 7.1 | Пример работы с операторами & и *

[1] «Undefined Behavior». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Undefined_behavior.

[2] «Common Weakness Enumeration». CWE. Был доступен: 15 сентября 2023 г. https://cwe.mitre.org/data/definitions/824.html.

[3] «Dangling Pointer». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Dangling_pointer.

7.4. Передача аргументов по указателю

В C++ есть три способа передачи аргументов функции:

- **передача по значению (pass-by-value)**;
- **передача по ссылке (pass-by-reference)**;
- **передача по указателю (pass-by-pointer)**, которую иногда называют «передачей по ссылке с аргументом-указателем».

В главе 5 мы рассмотрели первые два способа. Теперь мы объясним, как выполнять передачу по указателю. Подобно передаче по ссылке, передача по указателю позволяет изменять переменные в вызывающем коде, а также передавать в функцию большие объемы данных (большие объекты) без их копирования, то есть практически без нагрузки на компьютер. При вызове функции, получающей указатель, нужно передать ей адрес переменной, применив оператор взятия адреса (**&**) к имени этой переменной.

Пример передачи аргумента по значению

На илл. 7.2 и 7.3 вы видите два варианта программы с функциями, возводящими целое число в третью степень. На илл. 7.2 переменная `number` передается по значению (строка 12) в функцию `cubeByValue` (строки 17–19), которая возводит аргумент в третью степень и возвращает результат в `main` инструкцией `return` (строка 18)[1]. Результат сохраняется в переменной `number` (строка 12), заменяя ее исходное значение.

```
1 // fig07_02.cpp
2 // Передача переменной по значению для возведения в третью степень
3 #include <fmt/format.h>
4 #include <iostream>
5
6 int cubeByValue(int n); // Прототип функции
7
8 int main() {
9    int number{5};
10
11    std::cout << fmt::format("Original value of number is {}\n", number);
12    number = cubeByValue(number); // Передаем number в функцию cubeByValue
13    std::cout << fmt::format("New value of number is {}\n", number);
14 }
15
16 // Возводим в куб значение аргумента типа int и возвращаем результат
17 int cubeByValue(int n) {
18    return n * n * n; // Выполняем вычисление и возвращаем результат
19 }
```

```
Original value of number is 5
New value of number is 125
```

Илл. 7.2 | Передача переменной по значению для возведения в третью степень

[1] Мы могли бы возвести число n в третью степень выражением std::pow(n, 3).

Пример передачи аргумента по указателю

Пример на илл. 7.3 отличается тем, что переменная `number` передается функции `cubeByReference` по указателю (строка 13), то есть вызываемая функция получает в качестве аргумента не саму переменную, а указатель на нее: адрес переменной `number` в памяти компьютера. В функции `cubeByReference` (строки 18–20) для получения этого аргумента определен параметр `nPtr` (указатель на переменную типа `int`). Для возведения в третью степень того числа, на которое указывает `nPtr` (строка 19), эта функция использует разыменованный указатель `*nPtr` (псевдоним переменной `number` из `main`). Это вычисление напрямую изменяет значение переменной `number` в `main` (строка 10). Инструкцию в строке 19 можно сделать нагляднее, добавив круглые скобки:

```
*nPtr = (*nPtr) * (*nPtr) * (*nPtr); //  Возведение *nPtr в третью степень
```

```
 1 // fig07_03.cpp
 2 // Передача переменной по указателю
 3 // для возведения в третью степень
 4 #include <fmt/format.h>
 5 #include <iostream>
 6
 7 void cubeByReference(int* nPtr); // Прототип функции
 8
 9 int main() {
10    int number{5};
11
12    std::cout << fmt::format("Original value of number is {}\n", number);
13    cubeByReference(&number); // Передаем адрес переменной number
14    std::cout << fmt::format("New value of number is {}\n", number);
15 }
16
17 // Вычисляем третью степень *nPtr, напрямую изменяя переменную number
18 void cubeByReference(int* nPtr) {
19    *nPtr = *nPtr * *nPtr * *nPtr; // Возводим *nPtr в третью степень
20 }
```

```
Original value of number is 5
New value of number is 125
```

Илл. 7.3 | Передача переменной по указателю для возведения в третью степень

Любая функция, получающая в качестве аргумента адрес памяти (то есть указатель), должна определить для него параметр соответствующего типа. Например, заголовок функции `cubeByReference` (строка 18) определяет, что функция получает в качестве аргумента указатель на переменную `int`, сохраняет этот адрес в `nPtr` и не возвращает никакого значения.

При передаче по указателю фактически передается указатель по значению

Передача переменной по указателю реализована в C++ таким образом: переменная-указатель передается вызываемой функции по значению, то есть копируется в со-

ответствующий параметр функции (в параметр, предназначенный для получения указателя). Затем вызываемая функция может разыменовать указатель для прямого доступа к переменной вызывающего кода, что равнозначно передаче по ссылке.

Сравнительный анализ передачи по значению и передачи по указателю

На илл. 7.4 и 7.5 анализируются программы, показанные на илл. 7.2 и 7.3. Рядом с выражениями и именами переменных изображены прямоугольники со значениями, полученными на соответствующих шагах программ. В правом столбце каждой диаграммы показаны функции `cubeByValue` (илл. 7.2) и `cubeByReference` (илл. 7.3) на тех шагах, когда они выполняются.

7.5. Традиционные массивы в стиле C

Здесь мы рассмотрим традиционные массивы, унаследованные от языка C. Как и современные массивы `std::array`, они тоже являются структурами данных, имеющими фиксированный размер. Мы включили в книгу этот раздел, потому что вы встретите традиционные массивы в устаревшем коде.

Напомним, что при создании новых приложений надо, как правило, использовать современные массивы `std::array` и векторы `std::vector`, которые повышают надежность и безопасность приложений. Эта рекомендация обусловлена, в частности, тем, что объекты `std::array` и `std::vector` всегда «знают» (то есть хранят) свой собственный размер, в отличие от традиционных массивов. Если вы будете работать с кодом, в котором есть традиционные массивы, вы можете преобразовывать их в `std::array` с помощью функции `to_array` (раздел 7.6) или безопасно обрабатывать с помощью `span` (раздел 7.10). Иногда традиционные массивы по-прежнему необходимы, например, при обработке аргументов командной строки (раздел 7.11).

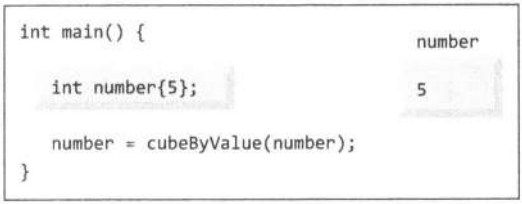
7.5.1. Объявление традиционного массива и доступ к его элементам

Объявление традиционного массива похоже по сути на объявление `std::array`, но отличается синтаксически. Чтобы объявить традиционный массив, вы должны указать тип и количество его элементов. Например, следующая инструкция объявляет массив из пяти элементов типа `int`:

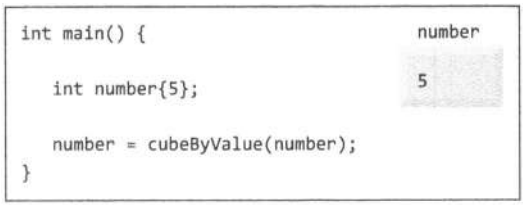
```
int c[5]; // c - массив из пяти элементов int
```

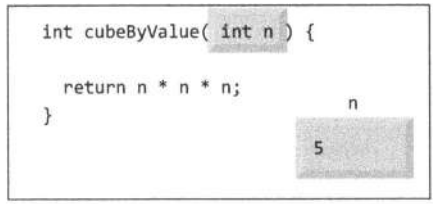
Для доступа к элементам традиционного массива используется оператор индекса (`[]`). В главе 6 мы отмечали, что он не проверяет границы массивов современного типа `std::array`. И с традиционными массивами этот оператор работает аналогично: проверка границ не выполняется автоматически, поэтому вы должны очень внимательно следить за правильностью индексов! Как вы помните, в `std::array` есть функция `at`, проверяющая границы. Традиционные массивы не имеют подобной функции, но для работы с ними вы можете использовать аналогичную функцию `gsl::at` из библиотеки `GSL` (Guidelines Support Library,

Шаг 1. Перед вызовом функции cubeByValue из `main`:

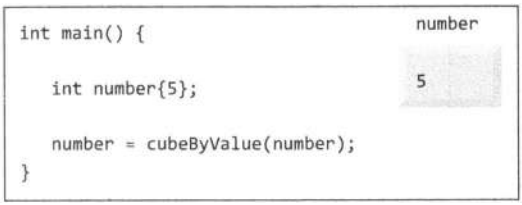


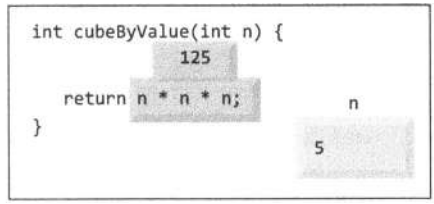
Шаг 2. После вызова функции cubeByValue:



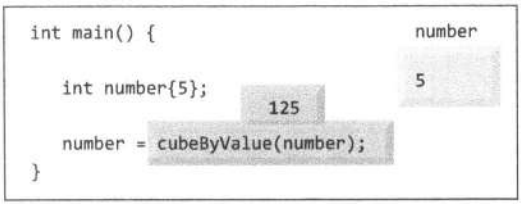


Шаг 3. После возведения в куб параметра n в функции cubeByValue, перед возвращением результата в `main`:

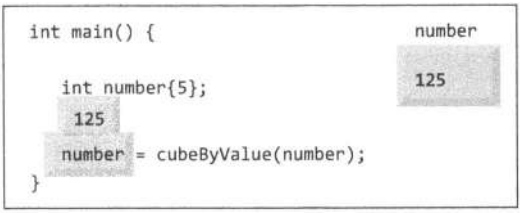




Шаг 4. После возвращения результата из cubeByValue в `main`, до сохранения результата в переменной number:

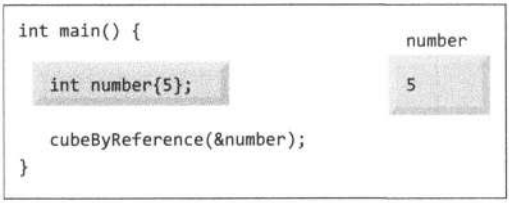


Шаг 5. После сохранения результата в переменной number:

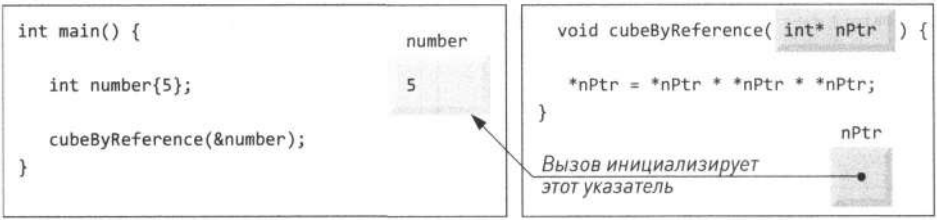


Илл. 7.4 | Анализ программы, показанной на илл. 7.2 (передача по значению)

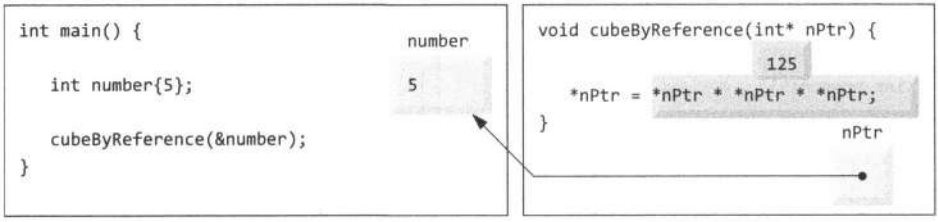
Шаг 1. Перед вызовом функции cubeByReference из main:



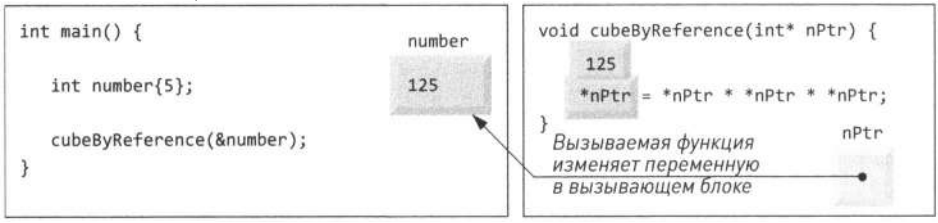
Шаг 2. После вызова функции cubeByReference, перед возведением в куб значения *nPtr:



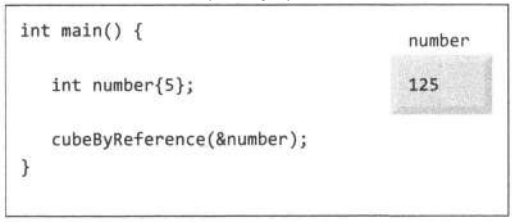
Шаг 3. Перед сохранением в *nPtr результата выражения 5 * 5 * 5:



Шаг 4. После сохранения в *nPtr числа 125, до возврата управления в main:



Шаг 5. После возврата управления из cubeByReference в main:



Илл. 7.5 | Анализ программы, показанной на илл. 7.3 (передача по указателю)

см. раздел 5.6.3), которая получает в качестве аргументов имя массива и индекс. Первым аргументом этой функции должно быть имя массива (а не указатель на первый элемент массива).

7.5.2. Инициализация традиционного массива

Вы можете инициализировать элементы традиционного массива списком инициализации, например:

```
int n[5]{50, 20, 30, 10, 40};
```

Эта инструкция создает и инициализирует массив из пяти целых чисел. Если количество инициализаторов в списке меньше, чем количество элементов массива, остальные элементы будут **инициализированы значением по умолчанию (value initialized)**: для основных числовых типов — `0`, для типа bool — `false`, для указателей — `nullptr`, а объекты будут инициализированы в соответствии с определениями их классов (мы расскажем об этом в главе 9). Если количество инициализаторов превысит размер массива, возникнет ошибка компиляции.

Компилятор умеет определять размер массива на основании списка инициализаторов. Следующее объявление создает массив из пяти элементов:

```
int n[]{50, 20, 30, 10, 40};
```

7.5.3. Передача традиционного массива функции

Когда традиционный массив передается функции, его имя неявно преобразуется в указатель на первый элемент массива — это называется **разложением массива (decaying to a pointer)**. Например, имя массива `n` из предыдущей инструкции неявно преобразуется в `&n[0]`, то есть в указатель на элемент со значением `50`. Чтобы передать адрес массива функции, вам не нужен оператор взятия адреса (`&`), вы просто передаете имя массива. Как было показано в разделе 7.4, функция, получающая указатель на переменную в вызывающем коде, может изменять эту переменную. Традиционный массив устроен аналогично: вызываемая функция может изменять его элементы прямо в вызывающем коде, если ее параметр, получающий массив в качестве аргумента, объявлен без квалификатора `const`, защищающего массив от изменений.

7.5.4. Объявление параметров функции, обрабатывающей традиционный массив

В заголовке функции, предназначенной для обработки традиционных массивов, обычно объявлены два параметра: для имени массива и для размера массива. Приведем пример:

```
int sumElements(const int values[], size_t numberOfElements)
```

Первым аргументом этой функции должен быть одномерный массив констант типа `int`. В отличие от `std::array` и `std::vector`, традиционный массив «не знает»

своего размера, поэтому вы должны сообщить его размер той функции, которую вызываете для его обработки.

Этот же заголовок можно написать и по-другому:

```
int sumElements(const int* values, size_t numberOfElements)
```

Компилятор не делает различий между функцией, получающей указатель, и функцией, получающей имя традиционного массива. В процессе компиляции (как говорят программисты, «под капотом» компилятора) фрагмент кода `const int values[]` преобразуется в `const int* values`. Поэтому функция должна каким-то образом «понять», что вы передали ей указатель не на единственную переменную, а на первый элемент массива.

Согласно C++ Core Guidelines, при создании новых программ не следует передавать традиционные массивы функциям по указателю[1]. Вместо этого надо передавать такие массивы через объекты класса `span` (C++20), потому что они хранят не только указатель на первый элемент массива, но и размер массива. Мы рассмотрим класс `span` в разделе 7.10, и вы увидите, что это превосходный способ передачи традиционных массивов функциям.

7.5.5. Пример с функциями sort, begin и end стандартной библиотеки (C++11)

В разделе 6.12 мы сортировали массив с именем `colors` (`std::array`), содержащий элементы типа `string`, таким образом:

```
std::sort(std::begin(colors), std::end(colors));
```

Здесь функции `begin` и `end` возвращают начало и конец массива, то есть сортируется весь массив `colors`. Функции `sort`, `begin` и `end` (а также многие другие функции стандартной библиотеки C++) могут применяться и к традиционным массивам. Например, отсортировать массив `n` (раздел 7.5.2) можно так:

```
std::sort(std::begin(n), std::end(n)); // Сортируем традиционный массив n
```

Если функции `begin` и `end` применяются к традиционному массиву, они работают только с его именем, а не с указателем на его первый элемент. Подчеркнем, что вы должны передавать традиционные массивы функциям, используя класс `span` (C++20), который мы рассмотрим в разделе 7.10.

7.5.6. Недостатки традиционных массивов

Традиционные массивы имеют несколько недостатков, ограничивающих их применение:

---

[1] C++ Core Guidelines, «I.13: Do Not Pass an Array as a Single Pointer». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-array.

- Их нельзя сравнивать операторами равенства и сравнения. Если вам надо сравнить два таких массива, придется по очереди сравнивать в цикле все их элементы. Если у вас есть два традиционных массива целых чисел с именами `array1` и `array2`, условие `array1 == array2` всегда будет ложным, даже если содержимое этих массивов идентично. Дело в том, что при таком сравнении выполняется разложение массивов, то есть их имена преобразуются в константные указатели на первые элементы. Разумеется, первые элементы сравниваемых массивов находятся в разных ячейках памяти.
- Невозможно присвоить сразу все элементы одного традиционного массива другому, выполнив одну простую операцию вроде `builtInArray1 = builtInArray2`.
- Они не знают собственного размера, поэтому любая функция, обрабатывающая традиционный массив, должна получать в качестве аргументов его имя и его размер.
- Они не выполняют автоматическую проверку границ! Каждое выражение, влияющее на обработку элементов традиционных массивов, вы должны составлять и проверять очень тщательно, чтобы ни один индекс не вышел за границу какого-либо массива.

7.6. Функция to_array (C++20), преобразующая традиционные массивы в std::array

Если вы работаете в сфере ИТ, вам иногда может понадобиться устаревший код C++, в котором есть традиционные массивы. В соответствии с рекомендациями C++ Core Guidelines вы должны по возможности использовать вместо традиционных массивов современные массивы `std::array` и векторы `std::vector`, которые существенно безопаснее и не превращаются в указатели при передаче функциям[1]. Функция `std::to_array` (C++20, заголовок `<array>`)[2] позволит вам легко создать современный массив `std::array` из традиционного массива или из списка инициализаторов. На илл. 7.6 показан пример работы с функцией `to_array`. Содержимое каждого массива `std::array` выводится на экран обобщенным лямбда-выражением (строки 9–15). Как вы уже знаете, если указать `auto` вместо типа параметра лямбды, компилятор автоматически определит тип на основе контекста, в котором появляется лямбда. Именно так в нашей программе определяется тип элементов обрабатываемого лямбдой массива `std::array`.

---

[1] C++ Core Guidelines, «SL.con.1: Prefer Using STL array or vector Instead of a C array». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rsl-arrays.

[2] Zhihao Yuan, «to_array from LFTS with Updates». 17 июля 2019 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p0325.

```
1 // fig07_06.cpp
2 // Создание массивов std::array с помощью функции to_array (C++20)
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <array>
6
7 int main() {
8    // Обобщенная лямбда, отображающая элементы массива
9    const auto display{
10      [](const auto& items) {
11         for (const auto& item : items) {
12            std::cout << fmt::format("{} ", item);
13         }
14      }
15   };
16
17   const int values1[]{10, 20, 30};
18
19   // Создаем std::array из традиционного массива
20   const auto array1{std::to_array(values1)};
21
22   std::cout << fmt::format("array1.size() = {}\n", array1.size())
23      << "array1: ";
24   display(array1); // Вызываем лямбду для отображения элементов
25
26   // Создаем std::array из списка инициализаторов
27   const auto array2{std::to_array({1, 2, 3, 4})};
28   std::cout << fmt::format("\narray2.size() = {}\n", array2.size())
29      << "array2: ";
30   display(array2); // Вызываем лямбду для отображения элементов
31
32   std::cout << '\n';
33 }
```

```
array1.size() = 3
array1: 10 20 30

array2.size() = 4
array2: 1 2 3 4
```

Илл. 7.6 | Создание массивов std::array с помощью функции to_array (C++20)

Создание std::array из традиционного массива

Строка 20 создает массив `std::array`, состоящий из трех элементов типа `int`, путем их копирования из традиционного массива `values1`. Мы используем ключевое слово `auto`, чтобы компилятор автоматически определил тип и размер `array1`. Если бы мы явно объявили тип и размер, то в случае их несоответствия возвращаемому значению функции `to_array` возникла бы ошибка компиляции. Мы присваиваем результат переменной `array1`. В строках 22 и 24 выводятся на экран размер и содержимое массива `std::array`, что подтверждает корректное преобразование.

Создание std::array из списка инициализаторов

В строке 27 показано, что функция `to_array` может создать массив `std::array` из списка инициализаторов. В строках 28 и 30 выводятся на экран размер и содержимое массива, что подтверждает его корректное создание.

7.7. Использование const с указателями и данными, на которые они указывают

В этом разделе мы расскажем, как использовать квалификатор `const` с указателями, чтобы соблюдать принцип наименьших привилегий. В главе 5 мы объяснили, как устроена передача по значению: значение аргумента копируется в параметр функции. Если эта копия изменится в вызываемой функции, исходное значение в вызывающем коде останется неизменным. Но иногда бывает нужно сделать так, чтобы даже копия в вызываемой функции не могла изменяться.

В тех случаях, когда какое-либо значение передается функции и не должно изменяться в ее теле, сделайте соответствующий параметр функции константным, то есть объявите его с квалификатором `const`. Перед использованием функции внимательно проверьте ее прототип, чтобы определить, какие параметры она должна изменять, а какие не должна.

Есть четыре способа объявления указателей:

- неконстантный указатель на неконстантные данные;
- неконстантный указатель на константные данные (илл. 7.7);
- константный указатель на неконстантные данные (илл. 7.8);
- константный указатель на константные данные (илл. 7.9).

Эти способы обеспечивают разные уровни привилегий доступа к данным.

7.7.1. Неконстантный указатель на неконстантные данные

Этот способ дает максимальные привилегии:

- данные можно изменять через разыменованный указатель;
- указатель можно изменять, чтобы он указывал на другие данные.

В объявлении такого указателя (например, `int* countPtr`) нет квалификатора `const`.

7.7.2. Неконстантный указатель на константные данные

Этот способ дает следующие привилегии:

- указатель можно изменять, чтобы он указывал на другие данные соответствующего типа;
- через указатель нельзя изменять данные.

Объявление такого указателя содержит квалификатор `const` слева от типа указателя[1]:

```
const int* countPtr;
```

Это объявление читается справа налево: «`countPtr` — указатель на константу типа `int`» или «`countPtr` — неконстантный указатель на целочисленную константу».

На илл. 7.7 показана ошибка компиляции, возникающая при попытке изменить константные данные с помощью неконстантного указателя.

```
1 // fig07_07.cpp
2 // Попытка изменить константные данные
3 // через неконстантный указатель
4
5 int main() {
6    int y{0};
7    const int* yPtr{&y};
8    *yPtr = 100; // Ошибка: нельзя изменить объект, объявленный константой
9 }
```

Сообщение компилятора GNU C++ об ошибке:

```
fig07_07.cpp: In function 'int main()':
fig07_07.cpp:8:10: error: assignment of read-only location '* yPtr'
    8 | *yPtr = 100; // error: cannot modify a const object
      | ~~~~~~^~~~~
```

Илл. 7.7 | Попытка изменить константные данные через неконстантный указатель

Всегда передавайте аргументы основных типов (`int`, `double` и т. д.) по значению, за исключением тех случаев, когда вызываемая функция должна напрямую изменять значения в вызывающем коде. Это еще один пример принципа наименьших привилегий. Если большие объекты не должны изменяться вызываемой функцией, передавайте их по указателю на константные данные или по ссылке на константные данные (лучше по ссылке). Это дает преимущество в быстродействии благодаря отсутствию операций копирования, выполняемых при передаче по значению. Передача больших объектов по ссылке на константные данные или по указателю на константные данные так же безопасна, как передача по значению.

7.7.3. Константный указатель на неконстантные данные

Этот способ дает следующие привилегии:

- указатель всегда указывает на определенный адрес памяти;
- данные, которые находятся по этому адресу, можно изменять через указатель.

[1] Некоторые программисты предпочитают записывать это так: int const* count Ptr; (читается справа налево: «countPtr — это указатель на константу типа int»).

Константные указатели (объявленные как `const`) надо инициализировать при объявлении. Если указатель является параметром функции, он инициализируется тем указателем, который передается функции. При каждом вызове функции этот параметр инициализируется заново.

На илл. 7.8 показана ошибка компиляции, возникающая при попытке изменить константный указатель. В строке 9 объявляется указатель `ptr` с типом `int* const`. Это объявление читается справа налево: «`ptr` — константный указатель на неконстантную переменную типа `int`». Указатель инициализируется адресом целочисленной переменной `x`. В строке 12 мы пытаемся присвоить указателю `ptr` адрес переменной `y`, но компилятор выдает сообщение об ошибке. Строка 11, в которой переменной `*ptr` присваивается значение 7, не вызывает ошибки. Неконстантную переменную, на которую ссылается `ptr`, можно изменять с помощью разыменованного указателя `ptr`, даже если сам указатель был объявлен как константа.

```
1 // fig07_08.cpp
2 // Попытка изменить константный указатель на неконстантные данные
3
4 int main() {
5    int x, y;
6
7    // ptr - константный указатель на значение int, которое можно изменять
8    // через ptr, но он всегда указывает на определенный адрес памяти
9    int* const ptr{&x}; // Константный указатель должен быть инициализирован
10
11   *ptr = 7; // Допустимо: *ptr не является константой
12   ptr = &y; // Ошибка: ptr - константа, ей нельзя присвоить другой адрес
13 }
```

Сообщение компилятора Microsoft Visual C++ об ошибке:

```
error C3892: 'ptr': you cannot assign to a variable that is const
```

Илл. 7.8 | Попытка изменить константный указатель на неконстантные данные

7.7.4. Константный указатель на константные данные

Этот способ дает минимальные привилегии:

- указатель всегда указывает на определенный адрес памяти;
- данные, которые находятся по этому адресу, нельзя изменять через указатель.

На илл. 7.9 приведен пример: переменная-указатель `ptr` объявляется в строке 12 с типом `const int* const`. Это объявление читается справа налево: «`ptr` — константный указатель на константу типа `int`». После исходного кода показаны сообщения компилятора Clang из Xcode: об ошибке при попытке изменить данные, на которые ссылается `ptr` (строка 16), и об ошибке при попытке изменить адрес, сохраненный в переменной-указателе (строка 17). В строке 14 нет ошибки, поскольку эта строка ничего не изменяет: ни указатель, ни данные, на которые он указывает.

```
1 // fig07_09.cpp
2 // Попытка изменить константный указатель на константные данные
3 #include <iostream>
4
5 int main() {
6    int x{5};
7    int y{6};
8
9    // ptr - константный указатель на константу int,
10   // он всегда указывает на определенный адрес памяти,
11   // и через этот указатель нельзя изменять расположенную там константу
12   const int* const ptr{&x};
13
14   std::cout << *ptr << '\n';
15
16   *ptr = 7; // Ошибка: *ptr - константа, ее значение нельзя изменить
17   ptr = &y; // Ошибка: ptr - константа, ей нельзя присвоить другой адрес
18 }
```

Сообщение компилятора Clang (из Xcode) об ошибке:

```
fig07_09.cpp:16:9: error: read-only variable is not assignable
   *ptr = 7; // error: *ptr is const; cannot assign new value
   ~~~~ ^
fig07_09.cpp:17:8: error: cannot assign to variable 'ptr' with const-qualified
type 'const int *const'
   ptr = &y; // error: ptr is const; cannot assign new address
   ~~~ ^
```

Илл. 7.9 | Попытка изменить константный указатель на константные данные

7.8. Оператор sizeof

Унарный оператор `sizeof` определяет **во время компиляции программы** размер своего операнда, то есть количество байтов, занимаемых им в памяти. Операндом может быть традиционный массив, тип данных, переменная или константа. При применении к имени массива (илл. 7.10[1], строка 13) оператор `sizeof` возвращает суммарное количество байтов массива в виде значения типа `size_t`. В операционной системе нашего компьютера, на котором мы компилировали этот код, каждая переменная типа `double` хранится в `8` байтах памяти. Массив `numbers` содержит `20` таких элементов (строка 10), поэтому занимает в памяти `160` байтов. При применении к указателю (строка 21) оператор `sizeof` возвращает его размер в байтах, в нашей системе это `4` байта.

[1] Это учебный пример, демонстрирующий работу оператора sizeof. Если вы используете инструменты статического анализа кода (например, C++ Core Guidelines Checker в Microsoft Visual Studio), вы получите предупреждения, поскольку не следует передавать функциям традиционные массивы.

```
// fig07_10.cpp
// Оператор sizeof возвращает количество байтов,
// занимаемых в памяти традиционным массивом
#include <fmt/format.h>
#include <iostream>

size_t getSize(double* ptr); // Прототип функции

int main() {
   double numbers[20]; // 20 переменных double (160 байтов в нашей системе)

   std::cout << fmt::format("Number of bytes in numbers is {}\n\n",
      sizeof(numbers));

   std::cout << fmt::format("Number of bytes returned by getSize is {}\n",
      getSize(numbers));
}

// Возвращаем размер ptr
size_t getSize(double* ptr) {
   return sizeof(ptr);
}
```

```
The number of bytes in the array is 160
The number of bytes returned by getSize is 4
```

Илл. 7.10 | Оператор sizeof возвращает количество байтов, занимаемых в памяти традиционным массивом

Определение размеров основных типов, традиционных массивов и указателей

На илл. 7.11 показано, как определять оператором `sizeof` размеры основных типов данных. Выходные данные программы мы получили с помощью компилятора Clang из Xcode. На других платформах выходные данные могут отличаться. Когда мы запускали эту программу в нашей версии Windows, размер типа `long` был равен `4` байтам, а размер типа `long long` — `8` байтам. В нашей версии macOS размеры обоих типов равны `8` байтам. В строках 7–15 пустой список инициализаторов `{}` (C++11) неявно инициализирует каждую переменную значением `0`[1].

```
// fig07_11.cpp
// Определение размеров стандартных типов данных оператором sizeof
#include <fmt/format.h>
#include <iostream>

int main() {
```

Окончание ⇨

[1] В строке 16 мы используем const, а не constexpr, чтобы предотвратить ошибку несоответствия типов на этапе компиляции. Имя традиционного массива (строка 15) разыменовывается в const int*, поэтому мы должны объявить ptr с этим типом.

```
 7    constexpr char c{}; // Переменная char
 8    constexpr short s{}; // Переменная short
 9    constexpr int i{}; // Переменная int
10    constexpr long l{}; // Переменная long
11    constexpr long long ll{}; // Переменная long long
12    constexpr float f{}; // Переменная float
13    constexpr double d{}; // Переменная double
14    constexpr long double ld{}; // Переменная long double
15    constexpr int array[20]{}; // Традиционный массив
16    const int* const ptr{array}; // Переменная int*
17
18    std::cout << fmt::format("sizeof c = {}\tsizeof(char) = {}\n",
19       sizeof c, sizeof(char));
20    std::cout << fmt::format("sizeof s = {}\tsizeof(short) = {}\n",
21       sizeof s, sizeof(short));
22    std::cout << fmt::format("sizeof i = {}\tsizeof(int) = {}\n",
23       sizeof i, sizeof(int));
24    std::cout << fmt::format("sizeof l = {}\tsizeof(long) = {}\n",
25       sizeof l, sizeof(long));
26    std::cout << fmt::format("sizeof ll = {}\tsizeof(long long) = {}\n",
27       sizeof ll, sizeof(long long));
28    std::cout << fmt::format("sizeof f = {}\tsizeof(float) = {}\n",
29       sizeof f, sizeof(float));
30    std::cout << fmt::format("sizeof d = {}\tsizeof(double) = {}\n",
31       sizeof d, sizeof(double));
32    std::cout << fmt::format(sizeof ld = {}\tsizeof(long double) = {}\n",
33       sizeof ld, sizeof(long double));
34    std::cout << fmt::format("sizeof array = {}\n", sizeof array);
35    std::cout << fmt::format("sizeof ptr = {}\n", sizeof ptr);
36 }
```

```
sizeof c = 1      sizeof(char) = 1
sizeof s = 2      sizeof(short) = 2
sizeof i = 4      sizeof(int) = 4
sizeof l = 8      sizeof(long) = 8
sizeof ll = 8     sizeof(long long) = 8
sizeof f = 4      sizeof(float) = 4
sizeof d = 8      sizeof(double) = 8
sizeof ld = 16    sizeof(long double) = 16
sizeof array = 80
sizeof ptr = 8
```

Илл. 7.11 | Определение размеров стандартных типов данных оператором sizeof

Количество байтов, занимаемых в памяти элементами конкретного типа данных, может варьироваться в зависимости от платформы и компилятора. Если вы будете писать программу, работоспособность которой зависит от размеров типов данных, рассмотрите возможность использования **целочисленных типов фиксированного размера (fixed-size integer types)** (C++11, заголовок `<cstdint>`). Их полный список приведен здесь:

https://en.cppreference.com/w/cpp/types/integer

Оператор `sizeof` можно применять к любому выражению и к любому имени типа. Если операндом является имя переменной или выражение, оператор `sizeof` возвращает размер соответствующего типа. Круглые скобки нужны только в том случае, если в качестве операнда указан тип (например, `int`). Если операндом является выражение, круглые скобки не нужны. Не забывайте, что `sizeof` — это оператор этапа компиляции, поэтому его операнд не будет вычисляться во время выполнения программы.

7.9. Выражения и арифметические операции с указателями

В C++ можно выполнять арифметические операции с указателями. В этом разделе мы рассмотрим операторы с операндами-указателями и примеры работы с ними.

Арифметические операции возможны только с указателями на элементы традиционных массивов. Такие операции вам почти наверняка встретятся в устаревшем коде. Но согласно рекомендациям C++ Core Guidelines, любой указатель должен указывать только на единичный объект (а не на массив)[1] и вы не должны включать в новый код **арифметические операции с указателями**, поскольку они значительно повышают риск ошибок[2]. Если вам нужно обработать традиционный массив, используйте класс `span` (C++20), которому посвящен раздел 7.10.

Перечислим допустимые арифметические операции с указателями:

- инкремент (`++`) и декремент (`--`);
- прибавление целого числа к указателю (`+` или `+=`) и вычитание целого числа из указателя (`-` или `-=`);
- вычитание одного указателя из другого указателя того же типа.

Результаты этих операций зависят от размера элементов массивов, на которые ссылаются указатели, то есть от количества байтов, занимаемых каждым элементом в памяти. Поэтому арифметические операции с указателями являются платформенно-зависимыми.

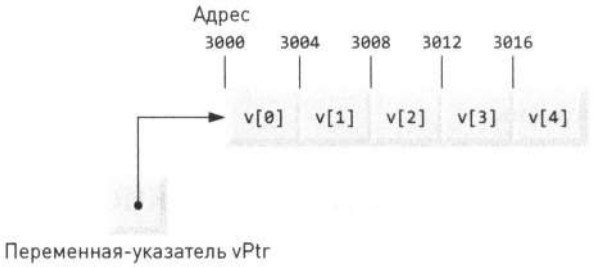
В большинстве современных настольных и мобильных компьютеров целые числа являются 4-байтовыми (32-разрядными) или 8-байтовыми (64-разрядными). Но среди миллиардов устройств «интернета вещей» (IoT) есть маломощные 8-разрядные и 16-разрядные компьютеры, в которых целые числа являются 2-байтовыми (согласно стандарту C++, это минимально возможный размер типа `int`). Предположим, что в вашей программе есть традиционный массив `int v[5]`, который начинается в памяти компьютера с адреса `3000`, а числа типа `int` на данном компьютере являются 4-байтовыми. Также предположим, что вы инициализиро-

---

[1] C++ Core Guidelines, «ES.42: Keep Use of Pointers Simple and Straightforward». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-ptr.

[2] C++ Core Guidelines «Pro.bounds: Bounds Safety Profile». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#SS-bounds.

вали указатель vPtr ссылкой на элемент v[0], то есть значение переменной vPtr равно 3000. Это состояние показано на схеме:



Поскольку имя традиционного массива неявно преобразуется в адрес его нулевого элемента, вы могли бы для получения этого состояния инициализировать переменную vPtr (чтобы она указывала на v) любой из двух инструкций:

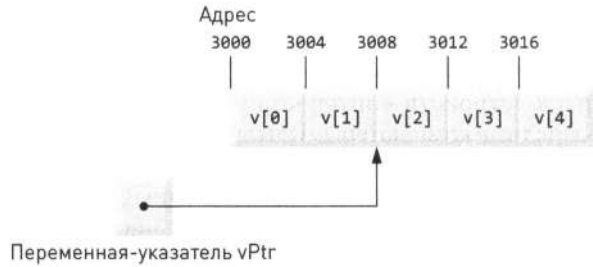
```
int* vPtr{v};
int* vPtr{&v[0]};
```

7.9.1. Прибавление и вычитание целых чисел

Теперь прибавим к vPtr число 2. В обычной арифметике выражение 3000 + 2 дает результат 3002. Но это не относится к арифметическим операциям с указателями. Если вы прибавите целое число к указателю (или вычтете целое число из указателя), указатель увеличится (или уменьшится) на это число, умноженное на размер типа данных, с которым работает указатель. Как вы уже знаете, размер типа данных зависит от платформы. В нашем примере инструкция:

```
vPtr += 2;
```

присвоит указателю vPtr значение 3008 (3000 + 2 * 4), поскольку каждое число типа int занимает в памяти 4 байта. Теперь vPtr указывает на элемент v[2]:



Теперь предположим, что значение vPtr равно 3016, то есть vPtr указывает на элемент v[4]. В этом случае выражение:

```
vPtr -= 4;
```

присвоит указателю `vPtr` значение `3000`, то есть вернет указатель к началу массива. Если вам нужно увеличивать или уменьшать указатель на `1`, то есть «передвигать» его вперед или назад на ближайший соседний элемент массива, вы можете делать это операторами инкремента (`++`) и декремента (`--`). Любое из выражений:

```
++vPtr;
vPtr++;
```

«передвигаст» указатель вперед, чтобы он указывал на следующий элемент массива. Любое из выражений:

```
--vPtr;
vPtr--;
```

«передвигает» указатель назад, чтобы он указывал на предыдущий элемент массива.

В арифметических операциях с указателями нет автоматической проверки границ массивов! Согласно C++ Core Guidelines, в современных программах вместо этих операций надо использовать класс `std::span` (раздел 7.10). Если вам все-таки придется выполнять операции сложения или вычитания с каким-либо указателем, вы должны быть на сто процентов уверены, что после каждой такой операции указатель остается в границах массива, то есть указывает на существующий элемент массива. Как вы скоро увидите, класс `std::span` выполняет проверку границ, что поможет вам избежать ошибок.

7.9.2. Вычитание одного указателя из другого

Указатели, указывающие на один и тот же традиционный массив, можно вычитать друг из друга. Например, если `vPtr` содержит адрес `3000`, а `v2Ptr` содержит адрес `3008`, инструкция:

```
x = v2Ptr - vPtr;
```

присвоит переменной `x` количество элементов традиционного массива от `vPtr` до `v2Ptr` — в нашем примере это `2`. Ни в коем случае нельзя предполагать, что две переменные одного и того же типа хранятся в памяти последовательно, если только они не являются смежными элементами традиционного массива. Вычитание или сравнение двух указателей, если они не указывают на элементы одного и того же традиционного массива, — это логическая ошибка.

7.9.3. Присваивание указателя

Указатель может быть присвоен другому указателю, если они оба имеют один и тот же тип[1]. Кроме того, любой указатель на основной тип или на тип класса можно присвоить указателю `void*` (то есть указателю на `void`) без приведения типа. Указатель `void*` может представлять любой тип указателя. Однако указатель типа

[1] Напомним, что константные указатели не могут быть изменены.

`void*` нельзя непосредственно присвоить указателю другого типа — сначала надо привести указатель типа `void*` к соответствующему типу с помощью `static_cast`.

7.9.4. Невозможность разыменования указателя void*

Указатель `void*` не может быть разыменован. Например, компилятор «знает», что `int*` указывает на 4-байтовый фрагмент памяти на компьютере с 4-байтовыми целыми числами. В этом случае разыменование `int*` создает значение *lvalue*, которое является псевдонимом для тех четырех байтов в памяти, которые хранят конкретное число типа `int`. Однако `void*` содержит адрес памяти для неизвестного типа данных. Вы не можете использовать `void*` в арифметических операциях и не можете его разыменовывать, потому что компилятору неизвестен тип (размер) данных, на которые он указывает.

Перечислим допустимые операции с указателями `void*`:

- сравнение указателя `void*` с другим указателем;
- приведение указателя `void*` к другому типу указателя;
- преобразование другого типа указателя в указатель `void*`.

Все остальные операции с указателями `void*` приведут к ошибкам компиляции.

7.9.5. Сравнение указателей

Указатели можно сравнивать операторами равенства и сравнения. Но никакие сравнения (`<`, `<=`, `>` и `>=`) не имеют смысла, если указатели относятся к разным традиционным массивам. Сравниваемые указатели должны указывать на один и тот же традиционный массив. При сравнении указателей сравниваются адреса памяти, хранящиеся в указателях. Например, сравнением двух указателей, указывающих на один и тот же массив, можно определить, что один из них указывает на элемент с большим индексом, чем другой. Операции сравнения часто нужны для того, чтобы выяснить, не является ли указатель нулевым (содержащим значение `nullptr`).

7.10. Готовые объекты: доступ к контейнерам через span (C++20)

Этот раздел посвящен работе с объектами класса `span` (C++20). Класс `span` (заголовок `<span>`) позволяет функциям обрабатывать элементы контейнеров: традиционных массивов, современных массивов `std::array` и векторов `std::vector`. Объект класса `span` является **представлением (view)** контейнера. Он дает обрабатывающей функции доступ к элементам контейнера, но не имеет своей копии этих элементов.

Мы уже обсуждали, как традиционный массив превращается в указатель, когда его имя передается функции. При этом, в частности, теряется информация о размере массива (несмотря на то, что вы указали размер, когда объявляли массив).

Вы убедились в этом на примере с оператором `sizeof` (илл. 7.10). В C++ Core Guidelines рекомендуется передавать традиционные массивы функциям через объекты `span`[1], содержащие не только указатель на первый элемент массива, но и размер массива. Следующая программа (илл. 7.12), разделенная на части для удобства обсуждения, демонстрирует важнейшие возможности объектов `span`.

```
1 // fig07_12.cpp
2 // Доступ к контейнерам через span (C++20)
3 #include <array>
4 #include <fmt/format.h>
5 #include <iostream>
6 #include <numeric>
7 #include <span>
8 #include <vector>
9
```

Илл. 7.12 | Доступ к контейнерам через span (C++20)

Функция displayArray

Для прямой передачи традиционного массива функции нужны два аргумента: имя массива и размер массива. Хотя параметр `items` (строка 12) объявляется с символами `[]`, на самом деле это просто указатель на значение типа `int`, который «не знает», сколько элементов содержит этот аргумент. Поэтому нужен второй аргумент. Такой подход вызывает ряд проблем. Например, код, вызывающий функцию `displayArray`, может случайно передать ей неправильное значение для параметра `size`. Тогда функция обработает не все элементы массива `items` или даже попытается получить доступ к элементу за границей массива, а это уже логическая ошибка и потенциальная опасность. Кроме того, у внешней итерации, показанной в строках 13–15, есть недостатки, которые мы рассмотрели ранее. Встроенный в Visual Studio инструмент C++ Core Guidelines Checker, проверяющий корректность кода, выдает предупреждения о функции `displayArray` и о передаче традиционных массивов функциям. Мы включили в этот пример функцию `displayArray` только для того, чтобы сравнить ее с функцией `displaySpan`, написанной правильно.

```
10 // Параметр items обрабатывается как const int*, поэтому в дополнение к
11 // нему надо передать размер массива, чтобы перебирать элементы в цикле
12 void displayArray(const int items[], size_t size) {
13    for (size_t i{0}; i < size; ++i) {
14       std::cout << fmt::format("{} ", items[i]);
15    }
16 }
17
```

[1] C++ Core Guidelines, «R.14: Avoid [] Parameters, Prefer span». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rr-ap.

Функция displaySpan

Согласно C++ Core Guidelines, в современном коде любой указатель должен указывать только на единичный объект[1] (а не на массив), а функции вроде `displayArray`, получающие указатель и размер массива, являются источником ошибок[2]. Поэтому надо передавать массивы функциям не напрямую, а через `span`. Наша функция `displaySpan` (строки 20–24) получает массив `items` через объект `span`, представляющий элементы массива как целочисленные константы (`const int`); мы сделали их константами, так как функция не должна их изменять.

```
18 // Параметр span содержит адрес первого элемента и количество элементов,
19 // поэтому мы можем перебирать их в цикле for для диапазона значений
20 void displaySpan(std::span<const int> items) {
21     for (const auto& item : items) { // Объекты span являются итерируемыми
22         std::cout << fmt::format("{} ", item);
23     }
24 }
25
```

Класс `span` инкапсулирует указатель и значение типа `size_t`, представляющее количество элементов. Когда вы передаете функции `displaySpan` традиционный массив (или `std::array`, или `std::vector`), C++ неявно создает объект `span`, содержащий и указатель на первый элемент массива, и размер массива, который компилятор самостоятельно находит в объявлении массива. Этот `span` предоставляет функции элементы исходного массива, который вы передали в качестве аргумента. В C++ Core Guidelines рекомендуется передавать `span` по значению — это не менее эффективно, чем передавать указатель и размер по отдельности[3], как мы сделали в функции `displayArray`.

Подобно массивам и векторам, объекты `span` имеют много возможностей, среди которых обработка элементов в циклах `for` для диапазона значений. Поскольку объект `span` создается на основе определенного компилятором размера массива, заданного при объявлении массива, вы получаете гарантию, что такой цикл никогда не обратится к несуществующим элементам за границами массива. Это исключает риск переполнения буфера и других опасных проблем, свойственных функциям, похожим на `displayArray`.

Функция times2

Поскольку `span` — это представление существующего контейнера, изменение элементов `span` изменяет исходные данные, то есть элементы этого контейнера.

---

[1] C++ Core Guidelines, «ES.42: Keep Use of Pointers Simple And Straightforward». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-ptr.

[2] C++ Core Guidelines, «I.13: Do Not Pass an Array as a Single Pointer». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-array.

[3] C++ Core Guidelines, «F.24: Use a span<T> or a span_p<T> to Designate a Half-open Sequence». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-range.

Функция `times2` умножает на 2 каждый элемент массива `items` через `span<int>`. Обратите внимание, что мы используем неконстантную ссылку (без квалификатора `const`), чтобы вносить изменения в массив через `span`.

```
26 // Изменяем элементы массива через span
27 void times2(std::span<int> items) {
28     for (int& item : items) {
29         item *= 2;
30     }
31 }
32
```

Передача массива функции для отображения его элементов

В строках 34–36 объявляются: традиционный массив `values1`, массив `values2` (`std::array`) и вектор `values3` (`std::vector`). Каждый из них содержит пять элементов, хранящихся в памяти последовательно. В строке 41 вызывается функция `displayArray`, отображающая содержимое массива `values1`. Первый параметр функции `displayArray` — это указатель на `int`, поэтому мы не можем использовать имена контейнеров `std::array` и `std::vector` для передачи этих объектов функции `displayArray`.

```
33 int main() {
34     int values1[]{1, 2, 3, 4, 5};
35     std::array values2{6, 7, 8, 9, 10};
36     std::vector values3{11, 12, 13, 14, 15};
37
38     // Нужно указать размер, потому что параметр items
39     // является указателем на первый элемент аргумента
40     std::cout << "values1 via displayArray: ";
41     displayArray(values1, 5);
42
```

```
values1 via displayArray: 1 2 3 4 5
```

Неявное создание объекта span и его передача функции

В строке 46 вызывается функция `displaySpan` с аргументом `values1`. Параметр этой функции был объявлен так:

```
std::span<const int>
```

Поэтому компилятор создает объект `span`, содержащий `const int*` — указатель на первый элемент массива и `size_t` — размер массива, автоматически получаемый из объявления `values1` в строке 34. Поскольку `span` может представлять любую непрерывную последовательность элементов, мы также можем передать функции `displaySpan` контейнеры `std::array` и `std::vector` с элементами `int` (строки 50 и 52). Компилятор создаст соответствующие объекты `span`, содержащие указатель на первый элемент контейнера и размер контейнера. Это делает функцию

displaySpan более универсальной, чем displayArray, которая в этом примере может принимать только традиционные массивы.

```
43    // Компилятор знает размер массива values1 и автоматически создает span,
44    // содержащий указатель &values1[0] и размер массива
45    std::cout << "\nvalues1 via displaySpan: ";
46    displaySpan(values1);
47
48    // Компилятор также может создавать span для std::array и std::vector
49    std::cout << "\nvalues2 via displaySpan: ";
50    displaySpan(values2);
51    std::cout << "\nvalues3 via displaySpan: ";
52    displaySpan(values3);
53
```

```
values1 via displaySpan: 1 2 3 4 5
values2 via displaySpan: 6 7 8 9 10
values3 via displaySpan: 11 12 13 14 15
```

При изменении элементов объекта span изменяются элементы исходного контейнера

Напомним, что функция times2 умножает каждый элемент объекта span на 2. В строке 55 функция times2 вызывается с массивом values1 в качестве аргумента. Параметр этой функции был объявлен так:

```
std::span<int>
```

Поэтому компилятор создает span, содержащий int* — указатель на первый элемент массива и size_t — размер массива, автоматически получаемый из объявления values1 в строке 34. Чтобы показать, что функция times2 изменила данные исходного массива, в строке 57 мы выводим на экран обновленные значения values1. Как и displaySpan, функцию times2 можно вызывать и с другими контейнерами из этой программы: std::array и std::vector.

```
54    // Изменение исходных данных через span
55    times2(values1);
56    std::cout << "\n\nvalues1 after times2 modifies its span argument: ";
57    displaySpan(values1);
58
```

```
values1 after times2 modifies its span argument: 2 4 6 8 10
```

Явное создание объекта span и взаимодействие с ним

Вы можете явно создавать объекты span. В строке 60 создается span<int>, представляющий данные массива values1, — здесь компилятор использует CTAD для определения типа int по типу элементов values1. В строках 61–62 демонстрируются функции front и back, возвращающие первый и последний элементы объекта span, — в нашем случае это первый и последний элементы массива values1.

```
59    // В span есть целый ряд средств для работы с массивами и векторами
60    std::span mySpan{values1}; // span<int>
61    std::cout << "\n\nmySpan's first element: " << mySpan.front()
62       << "\nmySpan's last element: " << mySpan.back();
63
```

```
mySpan's first element: 2
mySpan's last element: 10
```

Философия C++ Core Guidelines заключается в «предпочтительной проверке кода во время компиляции, а не во время выполнения»[1]. Это позволяет компилятору находить многие ошибки и сообщать о них во время компиляции, поэтому вам не приходится писать дополнительный код, предотвращающий ошибки во время выполнения. В строке 60 компилятор определяет количество элементов в `span` (5) по объявлению массива `values1` в строке 34. Вы также можете явно указать тип и количество элементов при создании объекта `span`:

```
span<int, 5> mySpan{values1};
```

Компилятор проверит эту инструкцию и убедится, что вы создали `span` в соответствии с `values1`; в противном случае он сообщит об ошибке компиляции.

Использование span с алгоритмом accumulate стандартной библиотеки

Как показано в нашем примере, объекты `span` являются итерируемыми. Это значит, что вы можете использовать функции `begin` и `end` с объектами `span`, чтобы передавать их алгоритмам (функциям, выполняющим алгоритмические операции) стандартной библиотеки C++, в том числе функциям `accumulate` (строка 66) и `sort`. Мы рассмотрим эти алгоритмы в главе 14.

```
64    // Объекты span совместимы с алгоритмами стандартной библиотеки C++
65    std::cout << "\n\nSum of mySpan's elements: "
66       << std::accumulate(std::begin(mySpan), std::end(mySpan), 0);
67
```

```
Sum of mySpan's elements: 30
```

Создание вложенных представлений

Напомним, что объекты `span` являются **представлениями (views)** контейнеров. В классе `span` есть функции `first`, `last` и `subspan`, позволяющие создавать **вложенные представления (subviews)**, представляющие не весь контейнер, а его фрагменты. В строках 70 и 72 мы используем функции `first` и `last` для создания вложенных представлений, содержащих первые три и последние три элемента массива `values1` соответственно. В строке 74 мы используем функцию `subspan` для получения вложенного представления, содержащего 3 элемента, начиная

[1] C++ Core Guidelines, «P.5: Prefer Compile-Time Checking to Run-Time Checking». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rpcompile-time.

с индекса 1. В каждом случае мы передаем вложенные представления функции displaySpan, чтобы показать корректность их создания.

```
68    // Класс span позволяет создавать вложенные представления контейнеров
69    std::cout << "\n\nFirst three elements of mySpan: ";
70    displaySpan(mySpan.first(3));
71    std::cout << "\nLast three elements of mySpan: ";
72    displaySpan(mySpan.last(3));
73    std::cout << "\nMiddle three elements of mySpan: ";
74    displaySpan(mySpan.subspan(1, 3));
75
```

```
First three elements of mySpan: 2 4 6
Last three elements of mySpan: 6 8 10
Middle three elements of mySpan: 4 6 8
```

При изменении элементов вложенных представлений изменяются элементы исходного контейнера

Через вложенные представления неконстантных данных (объявленных без квалификатора const) можно изменять эти данные. В строке 77 мы передаем функции times2 объект span, представляющий 3 элемента массива values1, начиная с индекса 1. В строке 79 мы выводим на экран обновленные значения элементов массива values1, чтобы подтвердить их изменение.

```
76    // Изменение вложенных представлений изменяет исходные данные
77    times2(mySpan.subspan(1, 3));
78    std::cout << "\n\nvalues1 after modifying elements via span: ";
79    displaySpan(values1);
80
```

```
values1 after modifying elements via span: 2 8 12 16 10
```

Доступ к элементам span с помощью оператора []

Напомним, что оператор [] дает доступ к элементам традиционных массивов, современных массивов std::array и векторов std::vector, не выполняя при этом проверку границ. Этот оператор можно использовать и для доступа к элементам объектов span, и тоже без проверки границ. В строке 82 мы выводим на экран элемент с индексом 2[1].

```
81    // Доступ к элементу span через оператор []
82    std::cout << "\n\nThe element at index 2 is: " << mySpan[2];
83 }
```

```
The element at index 2 is: 12
```

[1] В классе std::span нет функции at, проверяющей границы, но ее могут добавить в будущем.

7.11. Строки на основе указателей

Вы уже работали с классом `string` стандартной библиотеки C++, представляющим строки текста в виде полноценных объектов. В главе 8 класс `std::string` рассматривается подробнее. А в этом разделе мы кратко расскажем о **строках в стиле C (C-strings)**, которые основаны на указателях.

Конечно, строки класса `std::string` предпочтительнее, поскольку этот класс позволяет избежать многих проблем безопасности и ошибок, свойственных операциям со строками в стиле C. Но в некоторых случаях строки в стиле C необходимы, например при обработке аргументов командной строки. Кроме того, если вы будете работать с устаревшим кодом, написанным на языках C и C++, вы почти наверняка найдете там строки в стиле C.

Символы и символьные константы

Символы — это те «кирпичики», из которых строка за строкой составляется исходный код на языке C++. Исходный код каждой программы — это множество символов, которые при осмысленной группировке интерпретируются компилятором как инструкции и данные, используемые для выполнения задач. Исходный код может содержать **символьные константы (character constants)** типа `char`. Каждая символьная константа — это целое число, которое записывается в исходном коде в виде символа, заключенного в одинарные кавычки. Значение такой константы — это номер ее символа в наборе символов (в кодировке) операционной системы компьютера. Например, константа `'z'` имеет значение `122`, соответствующее букве `z` в наборе символов ASCII (см. приложение Б). Или, например, константа `'\n'` имеет значение `10`, соответствующее символу «перевод строки» в наборе символов ASCII.

Строка в стиле C — это традиционный массив

Строка в стиле C — это фактически традиционный массив символов, заканчивающийся **нулевым символом (null character)** `'\0'`, который обозначает конец строки в памяти. Доступ к строке в стиле C осуществляется по указателю на ее первый символ (независимо от длины строки).

Строковые литералы в качестве инициализаторов

Строковыми литералами можно инициализировать традиционные массивы символов (массивы элементов типа `char`) и переменные типа `const char*`. Например, каждое из этих объявлений инициализирует переменную строкой `"blue"`:

```
char color[]{"blue"};
const char* colorPtr{"blue"};
```

Первое объявление создает традиционный массив `color` из пяти элементов, содержащий символы `'b'`, `'l'`, `'u'`, `'e'` и `'\0'`. Второе объявление создает переменную-указатель `colorPtr`, указывающую на букву `b` в строке `"blue"` (которая заканчивается нулевым символом `'\0'`) где-то в памяти. Первое объявление можно реализовать

и другим способом: с помощью состоящего из отдельных символов списка инициализаторов, в который добавлен завершающий символ '\0', например:

```
char color[]{'b', 'l', 'u', 'e', '\0'};
```

Строковые литералы существуют в программе постоянно: с момента запуска до завершения. К ним можно получать доступ из любой точки программы. Строковые литералы неизменяемы: после объявления никакие изменения в них невозможны.

Проблемы строк в стиле C

Если в традиционном массиве элементов char не выделить достаточно места для хранения символа '\0', который должен завершать строку в стиле C, вы получите логическую ошибку. Создание или использование строки в стиле C, не заканчивающейся символом '\0', может привести к логическим ошибкам.

Сохраняя строки в традиционном массиве элементов char, убедитесь, что его размер достаточно велик для самой длинной строки, которая в нем может быть когда-либо сохранена. C++ допускает строки любой длины. Но если строка длиннее традиционного массива, в который вы ее сохраняете, все не поместившиеся в массив символы будут записаны в последующие ячейки памяти. Это приведет к краху программы или к логическим ошибкам и опасным уязвимостям.

Отображение C-strings

Традиционный массив элементов char, содержащий строку с завершающим нулевым символом, можно выводить на экран с помощью cout и <<. Например, инструкция:

```
std::cout << color;
```

отображает традиционный массив color. Объект cout не учитывает размер традиционного массива. Символы выводятся до тех пор, пока не встретится завершающий нулевой символ, который не отображается. Объекты cin и cout обрабатывают массивы символов (элементов типа char) как строки, заканчивающиеся нулевыми символами. Но аналогичная обработка ввода и вывода не выполняется для традиционных массивов с элементами других типов.

7.11.1. Аргументы командной строки

В некоторых случаях традиционные массивы и строки в стиле C необходимы. Например, многие программы принимают при запуске **аргументы командной строки (command-line arguments)**, содержащие информацию о требуемой конфигурации, об именах подлежащих обработке файлов и т. д. Чтобы передать программе аргументы этим стандартным способом, вы перечисляете их после имени программы, запуская ее из командной строки. Например, в системе Windows есть программа dir, выводящая на экран содержимое папки. Вы можете запустить эту программу с аргументом /p:

```
dir /p
```

В этой команде аргумент `/p` означает, что надо делать паузу после каждого заполнения экрана. Аналогичную программу `ls` в системах Linux и macOS вы можете запустить, например, с аргументом `-la`:

```
ls -la
```

В этой команде аргумент `-la` означает, что надо выводить на экран не краткие, а детализированные сведения.

Программа получает аргументы командной строки в виде строк в стиле C, при этом первым аргументом является имя программы. Чтобы использовать полученные аргументы в виде современных строк `std::string` или других типов данных (`int`, `double` и т. д.), вы должны сначала преобразовать аргументы в эти типы. На илл. 7.13 показан код и пример запуска программы, которая выводит на экран количество переданных ей аргументов командной строки, а затем — каждый из них в отдельной строке.

```
1 // fig07_13.cpp
2 // Получение аргументов командной строки
3 #include <fmt/format.h>
4 #include <iostream>
5
6 int main(int argc, char* argv[]) {
7    std::cout << fmt::format("Number of arguments: {}\n\n", argc);
8
9    for (int i{0}; i < argc; ++i) {
10      std::cout << fmt::format("{}\n", argv[i]);
11   }
12 }
```

```
fig07_13 Amanda Green 97
Number of arguments: 4

fig07_13
Amanda
Green
97
```

Илл. 7.13 | Получение аргументов командной строки

Чтобы программа получила аргументы командной строки, функция `main` должна быть объявлена с двумя параметрами (строка 6), которые по соглашению имеют имена `argc` и `argv`. Первый аргумент — переменная `int`, представляющая количество аргументов. Второй аргумент — указатель `char*`, указывающий на первый элемент традиционного массива символов (некоторые программисты записывают этот аргумент так: `char** argv`). Первый элемент массива — это строка в стиле C, содержащая имя программы[1]. Остальные элементы представляют собой строки в стиле C, содержащие остальные аргументы командной строки.

[1] Стандарт C++ допускает, что на некоторых платформах argv[0] может быть пустой строкой.

Следующая команда запускает программу `fig07_13` и передает ей аргументы: `"Amanda"`, `"Green"` и `"97"`:

```
fig07_13 Amanda Green 97
```

В macOS и Linux вы должны запускать эту программу из текущего каталога: `"./fig07_13"`. В командной строке аргументы разделяются пробелами, а не запятыми. При выполнении этой команды функция `main` программы `fig07_13` получает количество аргументов (`4`) и 4-элементный массив строк в стиле C, в котором:

- элемент `argv[0]` содержит имя программы: `"fig07_13"` (или `"./fig07_13"` в macOS и Linux);
- элементы от `argv[1]` до `argv[3]` содержат строки `"Amanda"`, `"Green"` и `"97"` соответственно.

Вы сами решаете, как обрабатывать полученные аргументы в вашей программе. Из-за проблем, связанных со строками в стиле C, перед использованием аргументов командной строки преобразуйте их в `std::string`.

7.11.2. Еще одна возможность функции to_array (C++20)

В разделе 7.6 мы продемонстрировали преобразование традиционных массивов в `std::array` с помощью функции `to_array`. На илл. 7.14 показана еще одна возможность этой функции. Мы используем лямбда-выражение (строки 9–13), аналогичное выражению на илл. 7.6, для отображения содержимого `std::array` после вызова функции `to_array`.

```
1 // fig07_14.cpp
2 // Создание std::array из строкового литерала функцией to_array (C++20)
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <array>
6
7 int main() {
8    // Лямбда-выражение для отображения элементов items
9    const auto display{
10      [](const auto& items) {
11         for (const auto& item : items) {
12            std::cout << fmt::format("{} ", item);
13         }
14      }
15   };
16
17   // Инициализация std::array строковым литералом
18   // создает одноэлементный array<const char*>
19   const auto array1{std::array{"abc"}};
20   std::cout << fmt::format("array1.size() = {}\narray1: ",
21      array1.size());
22   display(array1); // Используем лямбду для отображения содержимого
23
```

```
24    // Создаем из строкового литерала массив символов
25    const auto array2{std::to_array("C++20")};
26    std::cout << fmt::format("\n\narray2.size() = {}\narray2: ",
27       array2.size());
28    display(array2); // Используем лямбду для отображения содержимого
29
30    std::cout << '\n';
31 }
```

```
array1.size() = 1
array1: abc

array2.size() = 6
array2: C + + 2 0
```

Илл. 7.14 | Создание std::array из строкового литерала функцией to_array (C++20)

Инициализация std::array строковым литералом создает одноэлементный массив

В строке 19 создается массив, в котором есть только один элемент `const char*`, указывающий на строку в стиле C `"abc"`.

Передача строкового литерала функции to_array создает массив символов std::array

При передаче строкового литерала функции `to_array` (строка 25) создается массив `std::array`, содержащий в виде отдельных элементов типа `char` все символы этого литерала и завершающий нулевой символ. В строках 26–27 подтверждается, что массив состоит из 6 элементов. В строке 28 выводится на экран содержимое массива. Нулевой символ не имеет визуального представления, поэтому его нет в выходных данных.

7.12. Другие темы, связанные с указателями

В последующих главах мы рассмотрим еще несколько тем, имеющих прямое отношение к указателям:

- В главе 10 мы будем использовать указатели с объектами классов, чтобы показать, что в операциях динамического полиморфизма (термин из объектно-ориентированного программирования) можно использовать ссылки или указатели, но ссылки предпочтительнее.
- В главе 11 мы рассмотрим управление динамически выделяемой памятью с помощью указателей, позволяющее вам по мере необходимости создавать и уничтожать объекты во время выполнения программы. Неправильное управление этим процессом является источником скрытых ошибок, в том числе утечек памяти. Мы покажем, как **умные указатели (smart pointers)** автоматически управляют памятью и другими ресурсами, которые надо возвращать операционной системе, когда они больше не нужны.

- В главе 14 мы покажем, что имя функции может быть рассмотрено как указатель на ее реализацию, а функции могут передаваться другим функциям по указателям.

7.13. Итоги

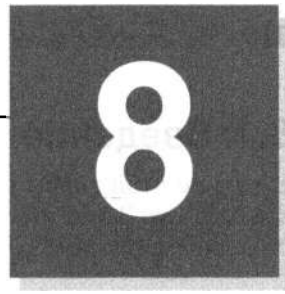
В этой главе рассматривались указатели, массивы на основе указателей (массивы в стиле C) и строки на основе указателей (строки в стиле C). При этом мы неоднократно акцентировали внимание на содержащиеся в C++ Core Guidelines рекомендации избегать указателей в большинстве случаев, заменяя их объектами `std::array`, `std::vector` и `std::string`.

Мы привели примеры объявления и инициализации указателей, а затем рассказали о необходимых для работы с ними операторах `&` и `*`. Мы сравнили два способа передачи аргументов функции: по ссылке и по указателю, пояснив, что вы должны, как правило, выбирать передачу по ссылке. Мы рассмотрели традиционные массивы и показали их тесную связь с указателями.

Мы обсудили сочетания квалификатора `const` с указателями и соответствующими данными. Мы использовали оператор `sizeof` для определения количества байтов, занимаемых в памяти данными основных типов и указателями. Мы продемонстрировали выражения и арифметические операции с указателями.

Наконец, мы кратко рассмотрели строки в стиле C и показали пример передачи в программу аргументов из командной строки. Для выполнения этой простой задачи в современном C++ по-прежнему необходимы строки и массивы в стиле C.

Надеемся, что после изучения этой главы вы сделали правильный вывод: программист должен избегать указателей, строк в стиле C и традиционных массивов во всех случаях, когда это возможно! При работе с устаревшим кодом, в котором есть массивы на основе указателей, вы можете преобразовывать их в современные массивы `std::array` функцией `to_array` (C++20) или безопасно обрабатывать через объекты `span` (C++20). В следующей главе мы рассмотрим основные операции со строками, предоставляемые классом `std::string`, а также начнем важную тему обработки файлов.



Строки и их представления, текстовые файлы, CSV-файлы и регулярные выражения

В этой главе вы:

- научитесь определять характеристики строк;
- научитесь искать, заменять и добавлять символы в строках;
- познакомитесь с функциями преобразования между строками и числами (C++11);
- познакомитесь с «облегченными» представлениями строк `string_view` (C++17);
- освоите операции чтения и записи текстовых файлов с последовательным доступом;
- узнаете о строковых потоках, которые удобны для проверки данных перед их обработкой;
- используя готовые объекты, прочитаете из CSV-файла набор данных «Титаник» и проанализируете часть этих данных;
- используя готовые объекты, начнете изучать регулярные выражения (C++11, заголовок `<regex>`), необходимые для поиска и замены фрагментов в тексте.

8.1. Введение

В начале этой главы вы узнаете о многих полезных возможностях класса `std::string`. Затем мы приступим к работе с представлениями `string_view`, текстовыми файлами, CSV-файлами и регулярными выражениями.

Операции со строками std::string

Начиная с главы 2, вы работали с текстовыми строками, которые являются объектами класса `std::string`. Теперь пришло время изучить более сложные операции с ними: присваивание, сравнение, создание и поиск подстрок, внесение всевозможных изменений. Кроме того, вы узнаете, как преобразовывать строки в числа и обратно.

Представления string_view (C++17)

Мы расскажем об объектах `string_view`, которые являются неизменяемыми представлениями строк `std::string` и строк в стиле C. Подобно объектам `std::span`, объекты `string_view` не содержат тех данных, которые они представляют. Вы увидите, что многие возможности объектов `string` и `string_view` похожи, поэтому представления хорошо подходят вместо строк в тех случаях, когда текст не должен изменяться.

Текстовые файлы и строковые потоки

Память компьютера (оперативная память) рассчитана на кратковременное хранение данных. Для длительного и надежного хранения данных предназначены **файлы (files)**. Компьютеры работают с файлами, которые находятся на локальных устройствах хранения (например, на флеш-накопителях) и «в облаке» (то есть на других компьютерах). Мы объясним, как написать на C++ программу, которая создает и обрабатывает текстовые файлы. Мы также покажем, как читать и записывать строки `std::string` через потоки `ostringstream` и `istringstream`.

Готовые объекты: CSV-файлы и набор данных о катастрофе «Титаника»

В этой главе есть два раздела о готовых объектах. В первом из них мы представим вам популярный файловый формат CSV. В этом формате часто хранят наборы данных, применяемые в работе с «большими данными» и в сфере искусственного интеллекта, например в системах обработки естественного языка и машинного обучения.

Начинающие специалисты по анализу и обработке данных часто практикуются на общедоступном наборе данных «Титаник». В нем перечислены все пассажиры и указано, кто из них выжил, когда пароход затонул 15 апреля 1912 года. Библиотека `rapidcsv` с открытым исходным кодом поможет вам прочесть набор данных «Титаник» из CSV-файла. Затем вы выполните простой анализ полученных данных.

Готовые объекты: регулярные выражения для поиска, замены и проверки фрагментов текста

Второй раздел о готовых объектах познакомит вас с регулярными выражениями. Они незаменимы в современных программах, обрабатывающих большие объемы текста. Вы будете составлять регулярные выражения в виде объектов `regex` и затем искать в тексте соответствующие им фрагменты с помощью функций из заголовка `<regex>`. В предыдущих главах наши программы не проверяли корректность ввода данных пользователем, но в профессиональном программном обеспечении такие проверки необходимы. Для этой цели программисты часто используют функции класса `std::string`, строковые потоки и регулярные выражения, о которых вы скоро узнаете.

8.2. Присваивание и конкатенация строк

На илл. 8.1 приведены примеры присваивания и конкатенации строк, являющихся объектами класса `std::string`.

```
// fig08_01.cpp
// Присваивание и конкатенация строк
#include <fmt/format.h>
#include <iostream>
#include <string>

int main() {
   std::string s1{"cat"};
   std::string s2; // Инициализируем пустой строкой
   std::string s3; // Инициализируем пустой строкой

   s2 = s1; // Присваиваем значение строки s1 строке s2
   s3.assign(s1); // Присваиваем значение строки s1 строке s3
   std::cout << fmt::format("s1: {}\ns2: {}\ns3: {}\n\n", s1, s2, s3);

   s2.at(0) = 'r'; // Изменяем строку s2
   s3.at(2) = 'r'; // Изменяем строку s3
   std::cout << fmt::format("After changes:\ns2: {}\ns3: {}", s2, s3);

   std::cout << "\n\nAfter concatenations:\n";
   std::string s4{s1 + "apult"}; // Выполняем конкатенацию
   s1.append("acomb"); // Создаем строку "catacomb"
   s3 += "pet"; // Создаем строку "carpet" перегруженным оператором +=
   std::cout << fmt::format("s1: {}\ns3: {}\ns4: {}\n", s1, s3, s4);

   // Добавляем символы с 4-й позиции до конца строки s1,
   // чтобы создать строку "comb" (строка s5 сначала была пустой)
   std::string s5; // Инициализируем пустой строкой
   s5.append(s1, 4, s1.size() - 4);
   std::cout << fmt::format("s5: {}", s5);
}
```

Окончание ⇨

```
s1: cat
s2: cat
s3: cat

After changes:
s2: rat
s3: car

After concatenations:
s1: catacomb
s3: carpet
s4: catapult
s5: comb
```

Илл. 8.1 | Присваивание и конкатенация строк

Присваивание строк

В строках 8–10 создаются объекты класса `std::string`: `s1`, `s2` и `s3`. Значение `s1` копируется в `s2` оператором присваивания (строка 12). Значение `s1` копируется в `s3` функцией `assign` класса `std::string` (строка 13). Эта версия функции `assign` выполняет такую же операцию, как оператор присваивания (`=`), но у функции `assign` есть еще целый ряд перегруженных версий, о которых вы можете узнать на этой странице:

https://en.cppreference.com/w/cpp/string/basic_string/assign

Например, одна из них копирует указанное количество символов:

```
target.assign(source, start, numberOfChars);
```

Здесь `source` — исходная строка (`std::string`) для копирования, `start` — начальный индекс, `numberOfChars` — количество копируемых символов.

Доступ к элементам строк по индексам

Функция `at` класса `std::string` (строки 16–17) присваивает значение `'r'` символу объекта `s2` с индексом `0` (образуя `"rat"`) и значение `'r'` символу объекта `s3` с индексом `2` (образуя `"car"`). Вы можете использовать функцию `at` и для того, чтобы получить из объекта `std::string` символ с указанным индексом. В этом случае (по аналогии с массивами `std::array` и векторами `std::vector`) функция `at` выполнит проверку границ и выдаст исключение `out_of_range`, если вы укажете недопустимый индекс. Оператор индекса `[]` работает иначе: он не проверяет границы объектов `std::string` (как и границы массивов `std::array` и векторов `std::vector`). Вы также можете перебирать все символы в объекте `std::string`, используя цикл `for` для диапазона значений:

```
for (char c : s3) {
   cout << c;
}
```

Этот способ исключает риск доступа к несуществующим символам за пределами объекта `std::string`.

Конкатенация строк

Объект `s4` инициализируется содержимым объекта `s1` с добавлением `"apult"` (строка 21). Применительно к объектам `std::string` оператор `+` выполняет конкатенацию строк. Функция `append` класса `std::string` выполняет конкатенацию `s1` и `"acomb"` (строка 22). Перегруженный оператор `+=` выполняет конкатенацию `s3` и `"pet"` (строка 23). Функция `append` выполняет конкатенацию пустого объекта `s5` и `"comb"` (строка 29), при этом фрагмент `"comb"` формируется ее аргументами: объектом `std::string`, из которого копируются символы (`s1`), начальным индексом (`4`) и количеством копируемых символов (`s1.size() - 4`).

8.3. Сравнение строк

Класс `std::string` содержит функции для сравнения своих объектов (илл. 8.2). В следующем примере мы несколько раз вызываем пользовательскую функцию `displayResult` (строки 7–17) для отображения результатов каждого сравнения.

```
 1 // fig08_02.cpp
 2 // Сравнение строк
 3 #include <fmt/format.h>
 4 #include <iostream>
 5 #include <string>
 6
 7 void displayResult(const std::string& s, int result) {
 8    if (result == 0) {
 9       std::cout << fmt::format("{} == 0\n", s);
10    }
11    else if (result > 0) {
12       std::cout << fmt::format("{} > 0\n", s);
13    }
14    else { // Результат < 0
15       std::cout << fmt::format("{} < 0\n", s);
16    }
17 }
18
19 int main() {
20    const std::string s1{"Testing the comparison functions."};
21    const std::string s2{"Hello"};
22    const std::string s3{"stinger"};
23    const std::string s4{s2}; // "Hello"
24
25    std::cout << fmt::format("s1: {}\ns2: {}\ns3: {}\ns4: {}",
26       s1, s2, s3, s4);
27
28    // Сравниваем s1 и s4
29    if (s1 > s4) {
```

Окончание ⇨

```
30       std::cout << "\n\ns1 > s4\n";
31    }
32
33    // Сравниваем s1 и s2
34    displayResult("s1.compare(s2)", s1.compare(s2));
35
36    // Сравниваем s1 (элементы 2-6) и s3 (элементы 0-4)
37    displayResult("s1.compare(2, 5, s3, 0, 5)",
38       s1.compare(2, 5, s3, 0, 5));
39
40    // Сравниваем s4 и s2
41    displayResult("s4.compare(0, s2.size(), s2)",
42       s4.compare(0, s2.size(), s2));
43
44    // Сравниваем s2 и s4
45    displayResult("s2.compare(0, 3, s4)", s2.compare(0, 3, s4));
46 }
```

```
s1: Testing the comparison functions.
s2: Hello
s3: stinger
s4: Hello

s1 > s4
s1.compare(s2) > 0
s1.compare(2, 5, s3, 0, 5) == 0
s4.compare(0, s2.size(), s2) == 0
s2.compare(0, 3, s4) < 0
```

Илл. 8.2 | Сравнение строк

Сравнение строк операторами равенства и сравнения

Объекты `std::string` можно сравнивать друг с другом и со строками в стиле C, используя операторы равенства и сравнения, возвращающие значение типа `bool`. Такие операции сравнения выполняются лексикографически, то есть по очереди сравниваются числовые значения всех символов в строках. Например, символ `'A'` имеет значение `65`, а символ `'a'` — `97` (см. приложение Б), поэтому строка `"Apple"` «меньше», чем `"apple"`, хотя эти слова могут показаться одинаковыми. В строке 29 объекты `s1` и `s4` сравниваются перегруженным оператором `>`. В данном случае строка `s1` начинается с заглавной буквы `'T'` (значение `84`), а строка `s4` начинается с заглавной буквы `'H'` (значение `72`), поэтому `s1` «больше», чем `s4`.

Сравнение строк функцией compare

В строке 34 функция `compare` класса `std::string` сравнивает `s1` и `s2`. Эта функция возвращает `0`, если `s1` и `s2` одинаковы; или положительное число, если `s1` лексикографически больше, чем `s2`; или отрицательное число, если `s1` лексикографически меньше, чем `s2`. Поскольку строка, начинающаяся с `'T'`, лексикографически боль-

ше строки, начинающейся с 'H', результатом сравнения является положительное число, что подтверждается выходными данными.

В строке 38 перегруженная функция compare сравнивает фрагменты строк s1 и s3. Первые два аргумента (2 и 5) — это начальный индекс и длина фрагмента строки s1 ("sting"). Третий аргумент — это строка для сравнения (s3). Последние два аргумента (0 и 5) — это начальный индекс и длина фрагмента строки s3 (тоже "sting"). Сравниваемые фрагменты одинаковы, поэтому функция возвращает 0, что подтверждается выходными данными.

В строке 42 другая перегруженная версия функции compare сравнивает фрагмент строки s4 со строкой s2. Первые два аргумента — это начальный индекс и длина фрагмента строки s4, а третий аргумент — строка для сравнения (s2). Сравниваемые фрагменты одинаковы, поэтому функция возвращает 0.

В строке 45 тем же способом выполняется сравнение первых трех символов строки s2 ("Hel") со строкой s4 ("Hello"). Фрагмент "Hel" совпадает с первыми тремя символами из "Hello", но он короче (меньше), чем "Hello", поэтому функция возвращает отрицательное число.

8.4. Создание подстроки

Функция substr класса std::string (илл. 8.3) возвращает фрагмент строки, или **подстроку (substring)**. Результатом является новый объект std::string, содержащий символы, скопированные из исходной строки. В следующем примере функция substr создает подстроку строки s, копируя фрагмент, который начинается с индекса 3 и состоит из 4 символов.

```
1 // fig08_03.cpp
2 // Создание подстроки
3 #include <iostream>
4 #include <string>
5
6 int main() {
7    const std::string s{"airplane"};
8    std::cout << s.substr(3, 4) << '\n'; // Получаем подстроку "plan"
9 }
```

```
plan
```

Илл. 8.3 | Создание подстроки

8.5. Обмен значениями двух строк

Класс std::string содержит функцию swap, выполняющую обмен значениями двух строк. На илл. 8.4 приведен пример: эта функция присваивает строке s1 прежнее значение s2, а строке s2 — прежнее значение s1.

```
1 // fig08_04.cpp
2 // Обмен значениями двух строк
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <string>
6
7 int main() {
8    std::string s1{"one"};
9    std::string s2{"two"};
10
11   std::cout << fmt::format("Before swap:\ns1: {}; s2: {}", s1, s2);
12   s1.swap(s2); // Выполняем обмен между s1 и s2
13   std::cout << fmt::format("\n\nAfter swap:\ns1: {}; s2: {}", s1, s2);
14 }
```

```
Before swap:
s1: one; s2: two

After swap:
s1: two; s2: one
```

Илл. 8.4 | Обмен значениями двух строк

8.6. Получение характеристик строки

Класс `std::string` содержит функции, которые предоставляют информацию о размере, емкости, максимальной длине и других характеристиках строки:

- **Размер (size)** строки — это количество символов, хранящихся в объекте `std::string` в данный момент.
- **Емкость (capacity)** строки — это количество символов, которое может быть сохранено в объекте `std::string`, прежде чем система должна будет выделить этому объекту дополнительную память для хранения большего количества символов. Выделение памяти для объектов `std::string` происходит автоматически. Емкость строки не может быть меньше размера строки, но может быть больше. Фактическая емкость в каждый момент времени зависит от реализации.
- **Максимальный размер (maximum size)** строки — это наибольшее количество символов, которое теоретически можно сохранить в объекте `std::string` на компьютере данного типа. Если это значение превышено, выдается исключение `length_error`[1].

В следующей программе, разделенной на части для удобства обсуждения (илл. 8.5), приведены примеры получения этих характеристик. Пользовательская функция `printStatistics` (строки 8–12) получает объект `std::string` и по очереди вы-

[1] В большинстве компьютерных систем максимальный размер настолько велик, что вы никогда не столкнетесь с исключением.

зывает четыре функции класса `std::string`, чтобы получить и вывести на экран характеристики строки: емкость (функция `capacity`), максимальный размер (функция `max_size`), размер (функция `size`) и результат проверки на пустую строку (функция `empty`).

```
1 // fig08_05.cpp
2 // Получение характеристик строки
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <string>
6
7 // Функция для вывода на экран
8 void printStatistics(const std::string& s) {
9    std::cout << fmt::format(
10       "capacity: {}\nmax size: {}\nsize: {}\nempty: {}",
11       s.capacity(), s.max_size(), s.size(), s.empty());
12 }
13
```

Илл. 8.5 | Получение характеристик строки

В строке 15 объявляется пустой объект `std::string` с именем `string1`, который затем передается функции `printStatistics` (строка 18). Эта функция возвращает размер `string1`, который равен `0`, поскольку строка не содержит символов. Для компиляторов Visual C++ и GNU g++ максимальный размер равен `9 223 372 036 854 775 807` (вы видите это число в выходных данных), а для компилятора Clang из Xcode он равен `18 446 744 073 709 551 599`. Объект `string1` является пустой строкой, поэтому функция `empty` возвращает `true`.

```
14 int main() {
15    std::string string1; // Пустая строка
16
17    std::cout << "Statistics before input:\n";
18    printStatistics(string1);
19
```

```
Statistics before input:
capacity: 15
max size: 9223372036854775807
size: 0
empty: true
```

Пользователь вводит текст (строка 21; мы ввели слово `tomato`), затем функция `printStatistics` (строка 24) отображает характеристики измененного объекта `string1`. Теперь размер строки равен `6`, она больше не является пустой.

```
20    std::cout << "\n\nEnter a string: ";
21    std::cin >> string1; // Ввод без пробелов
22    std::cout << fmt::format("The string entered was: {}\n", string1);
```

```
23    std::cout << "Statistics after input:\n";
24    printStatistics(string1);
25
```

```
Enter a string: tomato
The string entered was: tomato
Statistics after input:
capacity: 15
max size: 9223372036854775807
size: 6
empty: false
```

Пользователь вводит новый текст (строка 27; мы ввели слово `soup`), который сохраняется в `string1` вместо `"tomato"`. Функция `printStatistics` (строка 30) отображает характеристики измененного объекта `string1`. Как видите, теперь размер строки равен `4`.

```
26    std::cout << "\n\nEnter a string: ";
27    std::cin >> string1; // Ввод без пробелов
28    std::cout << fmt::format("The string entered was: {}\n", string1);
29    std::cout << "Statistics after input:\n";
30    printStatistics(string1);
31
```

```
Enter a string: soup
The string entered was: soup
Statistics after input:
capacity: 15
max size: 9223372036854775807
size: 4
empty: false
```

С помощью оператора `+=` (строка 33) выполняется конкатенация `string1` и строкового литерала, состоящего из `46` символов. Функция `printStatistics` (строка 36) снова отображает характеристики `string1`. Емкость `string1` была недостаточна для нового размера строки, поэтому система автоматически увеличила емкость до `63`. Размер `string1` теперь равен `50`. Алгоритм изменения емкости зависит от реализации.

```
32    // Добавляем 46 символов к string1
33    string1 += "1234567890abcdefghijklmnopqrstuvwxyz1234567890";
34    std::cout << fmt::format("\n\nstring1 is now: {}\n", string1);
35    std::cout << "Statistics after concatenation:\n";
36    printStatistics(string1);
37
```

```
string1 is now: soup1234567890abcdefghijklmnopqrstuvwxyz1234567890
Statistics after concatenation:
capacity: 63
max size: 9223372036854775807
size: 50
empty: false
```

Функция `resize` класса `std::string` (строка 38) вызывается, чтобы увеличить размер `string1` на `10` символов. Эти дополнительные элементы являются нулевыми символами. После этого функция `printStatistics` показывает, что емкость не изменилась (`63`), а размер равен `60`.

```
38    string1.resize(string1.size() + 10); // Добавляем 10 символов к string1
39    std::cout << "\n\nStatistics after resizing to add 10 characters:\n";
40    printStatistics(string1);
41    std::cout << '\n';
42 }
```

```
Statistics after resizing to add 10 characters:
capacity: 63
max size: 9223372036854775807
size: 60
empty: false
```

Обновление функции reserve в C++20

Вы можете изменить емкость строки без изменения размера, вызвав функцию `reserve` класса `std::string` с целочисленным аргументом. Если емкость меньше, чем значение аргумента, она увеличится до такого же или большего значения. Но если емкость больше, чем значение аргумента, она не уменьшится — так работает функция `reserve`, начиная с C++20. В более ранних версиях C++ эта функция может не только увеличивать, но и уменьшать емкость, причем если аргумент меньше размера строки, емкость уменьшается до размера строки.

8.7. Поиск подстрок и символов в строке

Класс `std::string` содержит функции для поиска в строках отдельных символов и последовательностей символов (подстрок, фрагментов). В следующей программе, разделенной на части для удобства обсуждения (илл. 8.6), приведены несколько примеров поиска. В строке 8 объявляется и инициализируется объект `std::string` с именем `s`.

```
 1 // fig08_06.cpp
 2 // Поиск подстрок и символов в строке
 3 #include <fmt/format.h>
 4 #include <iostream>
 5 #include <string>
 6
 7 int main() {
 8    const std::string s{"noon is 12pm; midnight is not"};
 9    std::cout << "Original string: " << s;
10
```

```
Original string: noon is 12pm; midnight is not
```

Илл. 8.6 | Поиск подстрок и символов в строке

Функции find и rfind

Затем дважды выполняется поиск подстроки "is" в строке s. Для этого вызываются функции find и rfind класса std::string (строки 12–13). Функция find выполняет поиск с начала строки (слева направо), а функция rfind — с конца (справа налево). Если подстрока найдена, каждая из функций возвращает индекс ее первого символа. В противном случае каждая из функций возвращает константу string::npos, означающую отсутствие искомой подстроки. Другие функции поиска, о которых мы расскажем в этом разделе, возвращают такие же значения, если не указано иное.

```
11    // Ищем фрагмент "is" с начала, а затем с конца строки s
12    std::cout << fmt::format("\ns.find(\"is\"): {}\ns.rfind(\"is\"): {}",
13       s.find("is"), s.rfind("is"));
14
```

```
s.find("is"): 5
s.rfind("is"): 23
```

Функция find_first_of

Функция find_first_of класса std::string (строка 16) ищет первое вхождение любого символа из подстроки "misop" в строке s. Эта функция выполняет поиск с начала строки (слева направо). Результатом поиска является символ 'o', имеющий в строке s индекс 1.

```
15    // Находим символ 'o' при поиске слева направо
16    int location{s.find_first_of("misop")};
17    std::cout << fmt::format("\ns.find_first_of(\"misop\") found {} at {}",
18       s.at(location), location);
19
```

```
s.find_first_of("misop") found o at 1
```

Функция find_last_of

Функция find_last_of класса std::string (строка 21) ищет последнее вхождение любого символа из подстроки "misop" в строке s. Эта функция выполняет поиск с конца строки (справа налево). Результатом поиска является символ 'o', имеющий в строке s индекс 27.

```
20    // Находим символ 'o' при поиске справа налево
21    location = s.find_last_of("misop");
22    std::cout << fmt::format("\ns.find_last_of(\"misop\") found {} at {}",
23       s.at(location), location);
24
```

```
s.find_last_of("misop") found o at 27
```

Функция find_first_not_of

Функция `find_first_not_of` класса `std::string` (строка 26) ищет в строке `s` первый символ из тех, которых нет в подстроке `"noi spm"`. Эта функция выполняет поиск с начала строки (слева направо). Результатом поиска является символ `'1'` с индексом `8`. Во втором вызове (строка 32) эта функция ищет в строке `s` первый символ из тех, которых нет в подстроке `"12noi spm"`. Результатом поиска является символ `';'` с индексом `12`. В третьем вызове (строка 38) эта функция ищет в строке `s` первый символ из тех, которых нет в подстроке `"noon is 12pm; midnight is not"`. Но в этой подстроке есть все символы из строки `s`. Поскольку символ не был найден, функция возвращает значение `string::npos` (в данном случае равное `-1`).

```
25    // Находим символ '1' при поиске слева направо
26    location = s.find_first_not_of("noi spm");
27    std::cout << fmt::format(
28       "\ns.find_first_not_of(\"noi spm\") found {} at {}",
29       s.at(location), location);
30
31    // Находим символ ';' с индексом 12
32    location = s.find_first_not_of("12noi spm");
33    std::cout << fmt::format(
34       "\ns.find_first_not_of(\"12noi spm\") found {} at {}",
35       s.at(location), location);
36
37    // Ищем символы, которых нет в "noon is 12pm; midnight is not"
38    location = s.find_first_not_of("noon is 12pm; midnight is not");
39    std::cout << fmt::format("\n{}: {}\n",
40       "s.find_first_not_of(\"noon is 12pm; midnight is not\")",
41       location);
42 }
```

```
s.find_first_not_of("noi spm") found 1 at 8
s.find_first_not_of("12noi spm") found ; at 12
s.find_first_not_of("noon is 12pm; midnight is not"): -1
```

8.8. Замена и удаление символов в строке

Класс `std::string` содержит функции для замены и удаления отдельных символов в строках. Этот пример, как и предыдущие, разделен на части (илл. 8.7). Объект `string1` объявляется с инициализацией строковыми литералами, разделенными символами пропуска (строки 9–13). Когда строковые литералы разделены таким образом, компилятор выполняет их конкатенацию, то есть объединяет в одну строку. Такая форма записи длинных строк может сделать ваш код более наглядным.

```
1 // fig08_07.cpp
2 // Замена и удаление символов в строке
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <string>
6
7 int main() {
8    // Компилятор объединяет все литералы в одну инициализирующую строку
9    std::string string1{"The values in any left subtree"
10       "\nare less than the value in the"
11       "\nparent node and the values in"
12       "\nany right subtree are greater"
13       "\nthan the value in the parent node"};
14
15   std::cout << fmt::format("Original string:\n{}\n\n", string1);
16
```

```
Original string:
The values in any left subtree
are less than the value in the
parent node and the values in
any right subtree are greater
than the value in the parent node
```

Илл. 8.7 | Замена и удаление символов в строке

Функция `erase` класса `std::string` (строка 17) удаляет символ с индексом `62` и все последующие символы до конца `string1`. При этом учитываются не только буквы, цифры и другие отображаемые символы, но и имеющиеся в `string1` символы перевода строки (`\n`), каждый из которых занимает одну позицию.

```
17   string1.erase(62); // Удаляем символ с индексом 62 и все последующие
18   std::cout << fmt::format("string1 after erase:\n{}\n\n", string1);
19
```

```
string1 after erase:
The values in any left subtree
are less than the value in the
```

В строках 20–26 функция `find` используется, чтобы найти все пробелы. Функция `replace` класса `std::string` заменяет каждый найденный пробел точкой. Эта версия функции `replace` принимает три аргумента:

- индекс символа, с которого должна начинаться замена;
- количество заменяемых символов;
- строка для замены.

Функция `find` возвращает `string::npos`, если искомый символ не найден. В строке 25 мы прибавляем `1` к переменной `position`, чтобы продолжить поиск с символа, следующего за найденным пробелом.

```
20    size_t position{string1.find(" ")}; // Находим первый пробел
21
22    // Заменяем каждый пробел точкой
23    while (position != std::string::npos) {
24       string1.replace(position, 1, ".");
25       position = string1.find(" ", position + 1);
26    }
27
28    std::cout << fmt::format("After first replacement:\n{}\n\n", string1);
29
```

```
After first replacement:
The.values.in.any.left.subtree
are.less.than.the.value.in.the
```

В строках 30–37 функция `find` используется, чтобы найти все точки. Функция `replace` заменяет каждую найденную точку, а также следующий за точкой символ двумя точками с запятой. Эта версия функции `replace` принимает пять аргументов:

- индекс символа, с которого должна начинаться замена;
- количество заменяемых символов;
- строка для замены (из которой будет взята заменяющая подстрока);
- индекс символа в строке для замены, с которого начинается заменяющая подстрока;
- количество символов в заменяющей подстроке.

```
30    position = string1.find("."); // Находим первую точку
31
32    // Заменяем каждую точку двумя точками с запятой
33    // Внимание: при этом перезаписываются другие символы
34    while (position != std::string::npos) {
35       string1.replace(position, 2, "xxxxx;;yyy", 5, 2);
36       position = string1.find(".", position + 2);
37    }
38
39    std::cout << fmt::format("After second replacement:\n{}\n", string1);
40 }
```

```
After second replacement:
The;;alues;;n;;ny;;eft;;ubtree
are;;ess;;han;;he;;alue;;n;;he
```

8.9. Вставка символов в строку

Класс `std::string` содержит перегруженные функции для вставки символов в строку (илл. 8.8). Функция `insert` (строка 14) вставляет фрагмент `"middle "`

перед символом строки s1 с индексом 10. Затем другая версия функции insert (строка 15) вставляет фрагмент "xx" перед символом строки s2 с индексом 3. Последние два аргумента определяют начало и конец той последовательности символов, которая должна быть вставлена в строку s2 из фрагмента "xx". Аргумент string::npos означает, что надо вставить весь этот фрагмент.

```
 1 // fig08_08.cpp
 2 // Вставка символов в строку
 3 #include <fmt/format.h>
 4 #include <iostream>
 5 #include <string>
 6
 7 int main() {
 8    std::string s1{"beginning end"};
 9    std::string s2{"12345678"};
10
11    std::cout << fmt::format("Initial strings:\ns1: {}\ns2: {}\n\n",
12       s1, s2);
13
14    s1.insert(10, "middle "); // Вставляем "middle " в позицию 10
15    s2.insert(3, "xx", 0, std::string::npos); // Вставляем "xx" в позицию 3
16
17    std::cout << fmt::format("Strings after insert:\ns1: {}\ns2: {}\n",
18       s1, s2);
19 }
```

```
Initial strings:
s1: beginning end
s2: 12345678

Strings after insert:
s1: beginning middle end
s2: 123xx45678
```

Илл. 8.8 | Вставка символов в строку

8.10. Преобразования между числами и строками (C++11)

В C++11 появились функции для преобразования числовых значений в строки и обратно.

Преобразование чисел в объекты std::string

Функция to_string из заголовка <string> возвращает строковое представление числового аргумента. Функция перегружена для основных числовых типов: int, unsigned int, long, unsigned long, long long, unsigned long long, float, double и long double.

Преобразование объектов std::string в числа

Для преобразования объектов `std::string` в числовые значения предназначены восемь функций из заголовка `<string>`:

Функция	Тип возвращаемого значения	Функция	Тип возвращаемого значения
Функции, преобразующие в целочисленные типы		Функции, преобразующие в типы с плавающей точкой	
`stoi`	`int`	`stof`	`float`
`stol`	`long`	`stod`	`double`
`stoul`	`unsigned long`	`stold`	`long double`
`stoll`	`long long`		
`stoull`	`unsigned long long`		

Каждая из этих функций пытается преобразовать начало своего аргумента типа `std::string` в числовое значение. Если это невозможно, функция выдает исключение `invalid_argument`. Если результат преобразования превышает диапазон возвращаемого функцией типа данных, функция выдает исключение `out_of_range`.

Функции, преобразующие строки в целые числа

Рассмотрим пример преобразования строки в целое число. Предположим, что у нас есть строка:

```
string s{"100hello"};
```

Тогда следующая инструкция преобразует начало строки в число `100` типа `int` и сохранит полученное значение в переменную `convertedInt`:

```
int convertedInt{stoi(s)};
```

Каждая функция, преобразующая строку в целочисленный тип, получает три аргумента (для последних двух есть значения по умолчанию):

- Строка `std::string`, содержащая символы для преобразования.
- Указатель на переменную типа `size_t`. В нее функция сохраняет индекс первого символа, который не был преобразован. Аргумент по умолчанию — `nullptr`, в этом случае функция не сохраняет индекс.
- Значение типа `int`, которое либо равно `0`, либо находится в диапазоне от `2` до `36` и обозначает основание системы счисления. Аргумент по умолчанию — `10`. Если вызвать функцию с аргументом `0`, она определит основание автоматически.

Таким образом, предыдущая инструкция эквивалентна следующей:

```
int convertedInt{stoi(s, nullptr, 10)};
```

Если добавить переменную типа `size_t` с именем `index`, инструкция

```
int convertedInt{stoi(s, &index, 2)};
```

воспримет последовательность символов `"100"` в начале строки `s` как число `100` в двоичной системе счисления (с основанием `2`), преобразует его в число `4` типа `int` (двоичное `100` — это десятичное `4`) и сохранит в переменной `index` индекс буквы `"h"` — первого символа, который не был преобразован.

Функции, преобразующие строки в числа с плавающей точкой

Каждая функция, преобразующая строки в числа с плавающей точкой, получает два аргумента:

- Строка, содержащая символы для преобразования.
- Указатель на переменную типа `size_t`. В нее функция сохраняет индекс первого символа, который не был преобразован (то есть количество преобразованных символов). Аргумент по умолчанию — `nullptr`, в этом случае функция не сохраняет индекс.

Рассмотрим преобразование следующей строки:

```
string s{"123.45hello"};
```

Инструкция

```
double convertedDouble{stod(s)};
```

преобразует начало строки `s` в число `123.45` типа `double` и сохранит его в переменную `convertedDouble`. Второй аргумент по умолчанию равен `nullptr`.

8.11. Представления string_view (C++17)

В C++17 появился тип данных `string_view` (заголовок `<string_view>`). Объекты этого типа являются **доступными только для чтения (read-only)** представлениями строк `std::string` или строк в стиле C. Кроме того, эти объекты могут быть представлениями диапазонов символов в контейнерах, например в массивах и векторах, состоящих из элементов типа `char`. Объект `string_view` так же, как и объект `std::span`, не владеет теми данными, которые представляет (в нем нет собственной копии этих данных). Он содержит:

- указатель на первый символ в непрерывной последовательности символов;
- количество символов.

Объект `string_view` не обновляется автоматически, если после его инициализации представляемые им данные (например, строка `std::string`) изменяются в размере!

С помощью представлений `string_view` многие из операций, доступных для строк `std::string`, можно применять и к строкам в стиле C, причем без необходимости

создания и инициализации объектов `std::string`, содержащих копии строк в стиле C. Согласно C++ Core Guidelines, вы должны отдавать предпочтение объектам `std::string` в тех случаях, когда вам нужно «владеть последовательностями символов», например вносить изменения в текст[1]. Но если какая-либо непрерывная последовательность символов нужна вам только для чтения, рекомендуется использовать `string_view`[2].

Работа с объектами string_view

Рассмотрим некоторые возможности `string_view`. Мы разделили этот пример на части для удобства обсуждения (илл. 8.9). В программу включен заголовок `<string_view>` (строка 6). Мы создаем объект `std::string` с именем `s1` (строка 9) и копируем его содержимое в объект `std::string` с именем `s2` (строка 10). Затем мы создаем объект `string_view` с именем `v1` и инициализируем его содержимым `s1` (строка 11).

```
 1 // fig08_09.cpp
 2 // Работа с представлениями string_view (C++17)
 3 #include <fmt/format.h>
 4 #include <iostream>
 5 #include <string>
 6 #include <string_view>
 7
 8 int main() {
 9    std::string s1{"red"};
10    std::string s2{s1};
11    std::string_view v1{s1}; // v1 "видит" содержимое строки s1
12    std::cout << fmt::format("s1: {}\ns2: {}\nv1: {}\n\n", s1, s2, v1);
13
```

```
s1: red
s2: red
v1: red
```

Илл. 8.9 | Работа с представлениями string_view

Объект string_view «видит» все изменения символов, которые он представляет

Поскольку объект `string_view` не содержит той последовательности символов, которую представляет, он «видит» все происходящие изменения исходных символов. Чтобы убедиться в этом, мы вносим изменение в `s1` (строка 15), а затем выводим на экран `s1`, `s2` и `v1`. Содержимое `s2` не изменилось, так как это независимая копия `s1`.

---

[1] C++ Core Guidelines, «SL.str.1: Use std::string to Own Character Sequences». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rstr-string.

[2] C++ Core Guidelines, «SL.str.2: Use std::string_view or gsl::span<char> to Refer to Character Sequences». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rstr-view.

```
14    // string_view "видит" все изменения исходной строки
15    s1.at(0) = 'R'; // Теперь s1 начинается с большой буквы
16    std::cout << fmt::format("s1: {}\ns2: {}\nv1: {}\n\n", s1, s2, v1);
17
```

```
s1: Red
s2: red
v1: Red
```

Сравнение объектов string_view

Подобно объектам `std::strings`, объекты `string_view` можно сравнивать между собой операторами равенства и сравнения. Кроме того, возможны сравнения между `string_view` и `std::string` (строки 19–20).

```
18    // Объекты string_view можно сравнивать между собой и с объектами string
19    std::cout << fmt::format("s1 == v1: {}\ns2 == v1: {}\n\n",
20       s1 == v1, s2 == v1);
21
```

```
s1 == v1: true
s2 == v1: false
```

Функции remove_prefix и remove_suffix

Вы можете легко удалить нужное количество символов с начала или конца `string_view`. Такие операции выполняются очень быстро, так как корректируется не сама строка, представленная объектом `string_view`, а только указатель на первый символ в последовательности (при удалении с начала) или на количество символов (при удалении с конца). В нашем примере функции `remove_prefix` и `remove_suffix` класса `string_view` (строки 23–24) вызываются для удаления первого и последнего символов соответственно. Как видите, символы удаляются только из представления `v1`, а исходная строка `s1` остается неизмененной.

```
22    // string_view позволяет удалить начало и конец строки
23    v1.remove_prefix(1); // Удаляем 1 символ с начала строки
24    v1.remove_suffix(1); // Удаляем 1 символ с конца строки
25    std::cout << fmt::format("s1: {}\nv1: {}\n\n", s1, v1);
26
```

```
s1: Red
v1: e
```

Обработка элементов string_view в цикле

В строке 28 создается объект `string_view` с именем `v2` и инициализируется строкой в стиле C. Объекты `string_view` являются итерируемыми, поэтому их элементы можно обрабатывать в цикле `for` для диапазона значений (строки 30–32).

```
27    // Объекты string_view - итерируемые
28    std::string_view v2{"C-string"};
29    std::cout << "The characters in v2 are: ";
30    for (char c : v2) {
31       std::cout << c << " ";
32    }
33
```

```
The characters in v2 are: C - s t r i n g
```

Операции с объектом string_view, представляющим строку в стиле C

Многие функции класса `string`, не изменяющие содержимое строки, определены и для `string_view`. Некоторые из них мы включили в нашу программу:

- Функция `size` возвращает количество символов, которые «видит» объект `string_view` (строка 35).
- Функция `find` возвращает индекс символа `'-'` (строка 36).
- Функция `starts_with` (C++20) возвращает ответ на вопрос, начинается ли `string_view` с символа `'C'` (строки 37–38).

Полный список функций класса `string_view` приведен здесь:

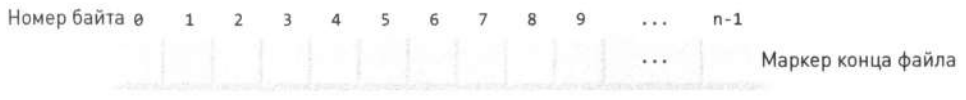
https://en.cppreference.com/w/cpp/string/basic_string_view

```
34    // Операции со строкой в стиле C через ее представление string_view
35    std::cout << fmt::format("\n\nv2.size(): {}\n", v2.size());
36    std::cout << fmt::format("v2.find('-'): {}\n", v2.find('-'));
37    std::cout << fmt::format("v2.starts_with('C'): {}\n",
38       v2.starts_with('C'));
39 }
```

```
v2.size(): 8
v2.find('-'): 1
v2.starts_with('C'): true
```

8.12. Файлы и потоки

C++ рассматривает каждый файл как последовательность байтов, например:



Текстовый файл обычно заканчивается либо **маркером конца файла (end-of-file marker)**, либо определенным байтом в соответствии с принятой в операционной системе структурой данных. При открытии файла создается объект, с которым связан поток. Объекты `cin`, `cout`, `cerr` и `clog`, предназначенные для работы с по-

токами, находятся в заголовке `<iostream>`. Потоки, связанные с этими объектами, обеспечивают каналы связи между вашей программой и конкретными файлами или устройствами. Через объекты `cin` (объекты стандартного потока ввода) программа получает данные от клавиатуры и от других устройств. Через объекты `cout` (объекты стандартного потока вывода) программа выводит данные на экран и на другие устройства. Через объекты `cerr` и `clog` (объекты стандартного потока ошибок) программа выводит сообщения об ошибках на экран и на другие устройства. Сообщения, записанные в `cerr`, выводятся немедленно, а сообщения, записанные в `clog`, хранятся в буфере памяти. Когда буфер заполняется или сбрасывается, его содержимое записывается в стандартный поток ошибок.

Файловые потоки

Для обработки файлов требуется заголовок `<fstream>`, который включает в себя следующие определения:

- `ifstream` — для чтения (ввода данных) из файлов;
- `ofstream` — для записи (вывода данных) в файлы;
- `fstream` — объединяет возможности `ifstream` и `ofstream`.

Возможности объектов `cout` и `cin`, которые мы рассматривали ранее, могут быть применимы и к файловым потокам.

8.13. Создание файла с последовательным доступом

C++ не требует какой-либо конкретной структуры файлов. Кроме того, не существует какой-либо обязательной концепции записей (элементов файла), содержащих взаимосвязанные элементы данных. Вы можете создать в файле практически любую структуру данных в соответствии с требованиями вашей программы. Мы покажем пример создания файла с очень простой структурой.

Программа (илл. 8.10) создает **файл с последовательным доступом (sequential file)**, хранящий данные учета денежных средств, причитающихся фирме от ее кредитных клиентов. Для каждого клиента в программу вводятся три значения: номер счета, имя и баланс (сумма, которую клиент должен компании за товары и услуги, полученные в прошлом). Эти три значения, взятые вместе, являются **записью (record)**, то есть набором данных о клиенте.

```
1 // fig08_10.cpp
2 // Создание последовательного файла
3 #include <cstdlib> // Содержит прототип функции exit
4 #include <fmt/format.h>
5 #include <fstream> // Содержит типы для обработки файловых потоков
6 #include <iostream>
7 #include <string>
8
```

```
 9 int main() {
10    // Объект ofstream открывает файл
11    if (std::ofstream output{"clients.txt", std::ios::out}) {
12       std::cout << "Enter the account, name, and balance.\n"
13          << "Enter end-of-file to end input.\n? ";
14
15       int account;
16       std::string name;
17       double balance;
18
19       // Вводим account, name и balance из cin, затем записываем в файл
20       while (std::cin >> account >> name >> balance) {
21          output << fmt::format("{} {} {}\n", account, name, balance);
22          std::cout << "? ";
23       }
24    }
25    else {
26       std::cerr << "File could not be opened\n";
27       std::exit(EXIT_FAILURE);
28    }
29 }
```

```
Enter the account, name, and balance.
Enter end-of-file to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z
```

Илл. 8.10 | Создание файла с последовательным доступом

Открытие файла

Чтобы записать данные в новый файл, мы прежде всего создаем объект `ofstream` (строка 11) и инициализируем его **именем файла (filename)** и **режимом открытия файла (file-open mode)**. Режим открытия файла `ios::out` предназначен для записи в файл (для вывода данных в файл). Этот режим установлен для всех объектов `ofstream` по умолчанию, поэтому второй аргумент в строке 11 необязателен. Открытие существующего файла в режиме `ios::out` приводит к немедленному удалению из этого файла всех данных без предупреждения! Если файл с указанным именем не существует, объект `ofstream` его создает.

Созданный нами объект `ofstream` с именем `output` связан с файлом `clients.txt` и открыт для записи. Мы не указали путь к файлу, поэтому он будет расположен в папке с исполняемым файлом программы. До C++11 имя файла указывалось в виде строки в стиле C. В C++11 была добавлена возможность указать имя файла в виде объекта `string`. В C++17 появился заголовок `<filesystem>`, содержащий средства управления файлами и папками. Таким образом, вы можете указывать открываемый файл в виде объекта `filesystem::path`.

В следующей таблице перечислены режимы открытия файлов. Эти режимы можно комбинировать, разделяя их оператором |:

Режим	Описание
`ios::app`	Добавление данных с конца файла, то есть вывод данных без изменения имеющихся в нем данных
`ios::ate`	Открытие файла для вывода и переход к концу файла. Используется для добавления данных в файл. Данные могут быть записаны в любом месте файла
`ios::in`	Открытие файла для чтения данных
`ios::out`	Открытие файла для записи данных
`ios::trunc`	Удаление содержимого файла. Это всегда происходит при открытии файла в режиме `ios::out`
`ios::binary`	Открытие файла для ввода или вывода двоичных (то есть нетекстовых) данных

Открытие файла с помощью функции open класса ofstream

Вы можете создать объект `ofstream` без немедленного открытия файла, а впоследствии связать с этим объектом нужный вам файл. Например, инструкция:

```
ofstream output;
```

создает объект `ofstream`, который не связан ни с каким файлом. После этого функция `open` открывает файл и связывает его с существующим объектом `ofstream`:

```
output.open("clients.txt", ios::out);
```

Напомним, что `ios::out` — это значение по умолчанию для второго аргумента.

Проверка успешности открытия файла

После создания объекта `output` и попытки открыть файл инструкция `if` использует этот объект в качестве условия (строка 11), чтобы определить, удалось ли открыть файл. Для этого к объекту файла неявно применяется перегруженная функция `operator bool` (C++11), вычисляющая его значение таким образом: `true`, если файл успешно открыт, или `false`, если возникла ошибка. Перечислим несколько причин, вызывающих ошибку при открытии файла:

- попытка открыть для чтения несуществующий файл;
- попытка открыть для чтения или для записи файл в папке, к которой у программы нет прав доступа;
- открытие файла для записи при отсутствии места на диске.

Если при проверке условия (строка 11) выясняется, что файл не удалось открыть, выводится сообщение об ошибке (строка 26), а затем вызывается функция `exit`

(строка 27) для завершения программы. Аргумент функции `exit` передается операционной системе, из которой была вызвана (запущена) программа. Передача значения `EXIT_SUCCESS` (определенного в `<cstdlib>`) сообщает операционной системе, что программа завершилась нормально; передача любого другого значения (в данном случае `EXIT_FAILURE`) сообщает, что программа завершилась из-за ошибки[1].

Обработка данных

Если файл успешно открыт (строка 11), программа начинает обработку данных. Пользователю предлагается заполнить поля в каждой записи о клиенте (строки 12–13) или ввести индикатор конца файла, если ввод данных завершен. Чтобы ввести индикатор конца файла в Linux или macOS, нажмите `Ctrl` + `d`. В Windows нажмите `Ctrl` + `z` и затем `Enter`.

При проверке условия в инструкции `while` (строка 20) к объекту `cin` неявно применяется функция `operator bool`, вычисляющая его значение таким образом: `true`, пока продолжаются успешные операции ввода данных, или `false`, если введен индикатор конца файла. Вы также можете вызвать функцию `eof` объекта `cin`, чтобы определить, был ли введен индикатор конца файла.

Строка 20 сохраняет каждый набор данных о клиенте в переменные `account`, `name` и `balance`, а также определяет, был ли введен индикатор конца файла. Если обнаружен конец файла (то есть пользователь нажал соответствующую комбинацию клавиш) или возникла ошибка операции ввода, `operator bool` возвращает `false` и инструкция `while` завершается.

В строке 21 все введенные данные записываются в файл `clients.txt` с помощью оператора вывода в поток (`<<`) и объекта `output`, который мы связали с файлом в начале программы. Записанные данные мы скоро прочитаем другой программой (раздел 8.14). Их также можно прочитать в любом текстовом редакторе, так как мы создали обычный текстовый файл.

Закрытие файла

Когда пользователь вводит индикатор конца файла, цикл `while` завершается. В этот момент достигается закрывающая фигурная скобка инструкции `if`, поэтому объект `output` выходит за пределы области видимости и файл автоматически закрывается. Вы всегда должны закрывать каждый файл, когда он больше не нужен в программе. Для этого можно явным образом закрыть связанный с файлом объект функцией `close`:

```
output.close();
```

---

[1] В функции main вы можете просто вернуть EXIT_SUCCESS или EXIT_FAILURE без вызова std::exit. Когда функция main завершается, ее локальные переменные уничтожаются, затем функция main вызывает std::exit, передавая значение из инструкции return в теле main (https://timsong-cpp.github.io/cppwp/n4861/basic.start.main#5). Напомним, что main неявно возвращает 0 в случае успешного выполнения, если в этой функции нет инструкции return.

Выходные данные

В конце илл. 8.10 в качестве примера приведены данные, которые мы ввели для пяти записей о клиентах. В последней строке вы видите символы ^Z, которые отображаются в Windows при вводе индикатора конца файла. Это диалоговое окно не показывает, каким образом введенные данные были записаны в файл. В следующем разделе мы рассмотрим программу для чтения и вывода на экран содержимого созданного файла.

8.14. Чтение данных из файла с последовательным доступом

Наша предыдущая программа (илл. 8.10) создала файл с последовательным доступом. Следующая программа (илл. 8.11) последовательно читает данные из файла `clients.txt` и выводит на экран все записи о клиентах. При создании объекта `ifstream` связанный с ним файл открывается для чтения (для ввода данных из файла). Объект `ifstream` инициализируется именем файла и режимом открытия файла. Мы создаем объект `ifstream` с именем `input` (строка 11), который открывает файл `clients.txt` для чтения[1]. Когда вам не нужно изменять содержимое файла, указывайте режим `ios::in`, чтобы открывать файл только для чтения и тем самым защитить его от случайной записи.

```
 1 // fig08_11.cpp
 2 // Чтение файла с последовательным доступом и вывод данных на экран
 3 #include <cstdlib>
 4 #include <fmt/format.h>
 5 #include <fstream> // Для работы с файлами
 6 #include <iostream>
 7 #include <string>
 8
 9 int main() {
10    // Объект ifstream открывает файл
11    if (std::ifstream input{"clients.txt", std::ios::in}) {
12       std::cout << fmt::format("{:<10}{:<13}{:>7}\n",
13          "Account", "Name", "Balance");
14
15       int account;
16       std::string name;
17       double balance;
18
19       // Выводим на экран каждую запись из файла
20       while (input >> account >> name >> balance) {
21          std::cout << fmt::format("{:<10}{:<13}{:>7.2f}\n",
22             account, name, balance);
23       }
24    }
```

[1] Файл clients.txt должен находиться в папке с исполняемым файлом программы.

```
25    else {
26       std::cerr << "File could not be opened\n";
27       std::exit(EXIT_FAILURE);
28    }
29 }
```

```
Account  Name      Balance
100      Jones       24.98
200      Doe        345.67
300      White        0.00
400      Stone      -42.16
500      Rich       224.62
```

Илл. 8.11 | Чтение файла с последовательным доступом и вывод данных на экран

Открытие файла для чтения

Все объекты класса `ifstream` по умолчанию открывают файлы для чтения, поэтому вы можете опустить второй аргумент (`ios::in`) при создании объекта `ifstream`:

```
std::ifstream input("clients.txt");
```

Вы можете создать объект `ifstream` без немедленного открытия файла, а впоследствии связать с этим объектом нужный вам файл. В строке 11 перед чтением данных из файла инструкция `if` использует объект `input` в качестве условия, чтобы определить, удалось ли открыть файл. Перегруженная функция `operator bool` вычисляет значение этого объекта таким образом: `true`, если файл успешно открыт, или `false`, если возникла ошибка.

Чтение данных из файла

В строке 20 считывается набор данных (запись) из файла. После первого выполнения этой инструкции переменная `account` содержит значение `100`, переменная `name` — `Jones`, а переменная `balance` — `24.98`. При каждом последующем выполнении этой инструкции в переменных `account`, `name` и `balance` сохраняется очередная запись о клиенте. В строках 21–22 каждая запись выводится на экран. Когда достигнут конец файла, неявный вызов `operator bool` в условии инструкции `while` возвращает `false`, объект `ifstream` выходит из области видимости (строка 24) и файл автоматически закрывается. Программа завершается.

Указатели позиции в файле

Многие программы последовательно читают все данные с начала файла до тех пор, пока не будут найдены нужные данные. Иногда требуется обработать файл несколько раз (с самого начала) во время выполнения программы. В классах `istream` и `ostream` есть функции `seekg` («seek get») и `seekp` («seek put»), которыми вы можете устанавливать **указатели позиции в файле (file-position pointers)**. Эти указатели содержат индексы тех байтов, которые должны быть прочитаны или записаны. Каждый объект `istream` имеет **указатель чтения (get pointer)** — индекс того байта в файле, с которого начнется следующая операция

чтения. Каждый объект `ostream` имеет **указатель записи (put pointer)** — индекс того байта в файле, с которого начнется следующая операция записи. Например, инструкция:

```
input.seekg(0);
```

устанавливает указатель чтения на начало файла (на байт с индексом `0`), связанного с объектом `input`. Аргументом этой функции должно быть целое число. Если ранее был введен индикатор конца файла, то для возобновления чтения из потока надо выполнить инструкцию:

```
input.clear();
```

Необязательный второй аргумент функции `seekg` определяет режим установки указателя:

- `ios::beg` (по умолчанию) — от начала потока;
- `ios::cur` — от текущей позиции в потоке;
- `ios::end` — назад от конца потока.

При установке указателя позиции относительно начала потока значением указателя является **смещение (offset)**, то есть количество байтов от начала файла. Приведем примеры установки указателя чтения на разные байты файла, связанного с объектом `fileObject`:

```
// Устанавливаем указатель на байт с индексом n (по умолчанию - ios::beg)
fileObject.seekg(n);
// Перемещаем указатель на n байт вперед от текущей позиции
fileObject.seekg(n, ios::cur);
// Устанавливаем указатель на n байт с конца файла
fileObject.seekg(n, ios::end);
// Устанавливаем указатель на последний байт файла
fileObject.seekg(0, ios::end);
```

Аналогичные операции выполняет в потоках `ostream` функция `seekp`. В классах `istream` и `ostream` также есть функции `tellg` и `tellp`, возвращающие текущие позиции указателей чтения и записи соответственно. Следующая инструкция присваивает значение указателя чтения переменной `location`, которая имеет тип `std::istream::pos_type`:

```
auto location{fileObject.tellg()};
```

8.15. Чтение и запись текста в кавычках (C++14)

Во многих текстовых файлах встречаются кавычки, в том числе универсальные двойные кавычки, применяемые программистами: `"C++20 for Programmers"`. Например, в текстовых файлах, представляющих собой веб-страницы HTML5, значения атрибутов заключены в кавычки. При создании таких программ, как

браузеры, отображающие содержимое веб-страниц, нужна возможность читать строки в кавычках и удалять эти кавычки.

Предположим, вам надо выполнить чтение набора данных из текстового файла (тем способом, который показан на илл. 8.11), но каждая запись о клиенте в этом файле имеет следующий формат с именем и фамилией:

```
100 "Janie Jones" 24.98
```

Напомним, что оператор извлечения из потока (>>) обрабатывает пробелы (символы пропуска) как разделители. Предположим, мы попробуем ввести эти данные, используя выражение из строки 20 на илл. 8.11:

```
input >> account >> name >> balance
```

Тогда переменной `account` будет присвоено значение `100` (первая операция ввода), а переменной `name` — значение `"Janie` (вторая операция ввода), причем открывающая кавычка станет частью строки. Третья операция ввода завершится ошибкой, поскольку третьим токеном (фрагментом данных) в потоке является последовательность символов `Jones"`, которая не может быть присвоена переменной `balance` типа `double`.

Чтение текста в кавычках

В C++14 был добавлен манипулятор потока `quoted` (заголовок `<iomanip>`) для чтения текста в кавычках из потока. Он включает в строки все пробелы, которые находятся во фрагментах, заключенных в кавычки, и удаляет кавычки. Снова рассмотрим ввод той же записи о клиенте:

```
100 "Janie Jones" 24.98
```

Инструкция

```
input >> account >> std::quoted(name) >> balance
```

присвоит переменной `account` значение `100`, переменной `name` — значение `Janie Jones` (одной строкой) и переменной `balance` — значение `24.98`. Если в данных, заключенных в кавычки, есть escape-последовательности \" или \\, они считываются как " или \ соответственно.

Запись текста в кавычках

Запись текста в кавычках происходит аналогичным образом. Например, если переменная `name` содержит значение `Janie Jones`, инструкция

```
outputStream << std::quoted(name);
```

запишет в `outputStream` строку

```
"Janie Jones"
```

Если строка содержит символы " или \, они будут записаны как \" или \\ соответственно.

8.16. Обновление данных в файле с последовательным доступом

Отформатированные данные, записанные в файл с последовательным доступом, как показано в разделе 8.13, нельзя изменять без риска уничтожения других данных в этом файле. Например, если в записи о клиенте надо заменить имя `White` именем `Worthington`, это невозможно сделать, не повредив файл. Предположим, в файле есть такая запись:

```
300 White 0.00
```

Если перезаписать ее в файле с той же позиции, но с более длинным именем клиента, запись будет выглядеть так:

```
300 Worthington 0.00
```

Эта новая запись на шесть символов длиннее прежней. Поэтому все символы после `"h"` в слове `"Worthington"` будут записаны в начало следующей записи, то есть повредят ее. Проблема модели форматированного ввода-вывода, использующей операторы ввода в поток (`<<`) и извлечения из потока (`>>`), заключается в том, что записи (элементы данных) могут различаться по размеру. Например, числа `7`, `14`, `-117`, `2074` и `27383` имеют тип `int`, поэтому занимают фиксированное количество байтов, если записываются в файл в формате своего типа. Но эти же числа потребуют разного количества байтов, если записываются в файл в виде форматированного текста (последовательностей символов). Поэтому модель форматированного ввода-вывода, как правило, не используется для обновления записей в файлах.

Для обновления записей в файлах с последовательным доступом есть относительно сложные способы. Например, чтобы заменить имя `White` более длинным именем `Worthington`, мы можем:

- скопировать в новый файл все записи, которые находятся перед записью `300 White 0.00`;
- внести в новый файл запись `300 Worthington 0.00`;
- скопировать в новый файл все оставшиеся записи.

Затем мы можем удалить старый файл и переименовать новый. Этот способ требует обработки всех имеющихся в файле записей, даже если обновляется только одна из них. Он может применяться, когда надо обновлять много записей за один проход.

8.17. Обработка строковых потоков

В дополнение к стандартным потокам ввода-вывода и файловым потокам ввода-вывода в C++ есть аналогичные потоки для обмена данными с находящимися в памяти строками класса `string`. Они называются **потоками ввода-вывода**

в памяти (in-memory I/O streams), или **строковыми потоками (string streams)** и определены в заголовке `<sstream>`. Вы можете читать данные из строки через строковый поток `istringstream` и записывать данные в строку через строковый поток `ostringstream`.

Шаблоны классов `istringstream` и `ostringstream` предоставляют такой же набор функций, как и классы `istream` и `ostream`, а также дополнительные функции, специфичные для форматирования в памяти. Объект `ostringstream` использует объект `string`, чтобы сохранять в него выводимые данные. Функция `str` класса `ostringstream` возвращает копию этого объекта `string`.

Строковые потоки применяются, в частности, для проверки данных. Программа может временно сохранить в объекте `string` последовательность символов, полученных из потока ввода (введенных с клавиатуры), а затем проанализировать эти данные и при необходимости скорректировать их (исправить формат). Затем программа может выполнить ввод данных из этого объекта `string`, поскольку теперь их формат наверняка соответствует требованиям.

Чтобы расширить возможности проверки данных, в C++11 были добавлены регулярные выражения, основанные на поиске и замене фрагментов текста по шаблонам. Если пользователь должен ввести телефонный номер в формате США, например `(800) 555-1212`, вы можете использовать регулярное выражение для проверки соответствия введенной строки этому формату. Многие веб-сайты предоставляют регулярные выражения для проверки адресов электронной почты, URL-адресов, телефонных номеров и других популярных видов данных. В разделе 8.20 мы расскажем о регулярных выражениях и приведем несколько примеров.

Вывод данных в ostringstream

В следующей программе (илл. 8.12) мы создаем объект `ostringstream` с именем `output` (строка 8), а затем записываем в него строки и числа, используя оператор вывода в поток.

```
1 // fig08_12.cpp
2 // Вывод данных в объект ostringstream
3 #include <iostream>
4 #include <sstream> // Заголовок для строковых потоков
5 #include <string>
6
7 int main() {
8    std::ostringstream output; // Создаем объект ostringstream
9
10   const std::string string1{"Output of several data types "};
11   const std::string string2{"to an ostringstream object:"};
12   const std::string string3{"\ndouble: "};
13   const std::string string4{"\nint: "};
14
15   constexpr double d{123.4567};
16   constexpr int i{22};
17
```

Окончание ⇨

```
18    // Выводим значения типов string, double и int в объект ostringstream
19    output << string1 << string2 << string3 << d << string4 << i;
20
21    // Вызываем функцию str, чтобы получить строку с содержимым ostringstream
22    std::cout << "output contains:\n" << output.str();
23
24    // Добавляем символы и снова вызываем функцию str
25    output << "\nmore characters added";
26    std::cout << "\n\noutput now contains:\n" << output.str() << '\n';
27 }
```

```
output contains:
Output of several data types to an ostringstream object:
double: 123.457
   int: 22

output now contains:
Output of several data types to an ostringstream object:
double: 123.457
   int: 22
more characters added
```

Илл. 8.12 | Вывод данных в объект ostringstream

В поток `output` поочередно выводятся (строка 19): строковые переменные `string1`, `string2`, `string3`, переменная `d` типа `double`, строковая переменная `string4` и переменная `i` типа `int`. Функция `output.str()` (строка 22) возвращает созданный объектом `output` объект `string`, который отображается на экране. Затем в поток `output` выводятся дополнительные данные (строка 25) и обновленное содержимое объекта `string` отображается на экране (строка 26).

Ввод данных из istringstream

Объект `istringstream` получает данные из находящейся в памяти строки, то есть из объекта типа `string`. Эти данные хранятся в объекте `istringstream` в виде символов. Ввод данных из объекта `istringstream` выполняется так же, как из любого файла. Конец строки интерпретируется объектом `istringstream` как конец файла.

На илл. 8.13 показан пример ввода данных из объекта `istringstream`. Объект `string` с именем `inputString` (строка 9) инициализируется как источник данных для двух строковых переменных (`"Amanda"` и `"test"`), для переменной типа `int` (`123`), для переменной типа `double` (`4.7`) и для переменной типа `char` (`'A'`). Объект `istringstream` с именем `input` (строка 10) инициализируется содержимым `inputString`. Выполняется ввод данных из `inputString` в переменные `s1`, `s2`, `i`, `d` и `c` (строка 17), значения которых отображаются на экране (строки 19–20). Затем мы снова пытаемся выполнить ввод из `input` (строка 23), но эта операция невозможна из-за отсутствия в `inputString` каких-либо дополнительных данных. Поэтому значение объекта `input` вычисляется как `false` и выполняется секция `else` инструкции `if...else`.

```
// fig08_13.cpp
// Ввод данных из объекта istringstream
#include <fmt/format.h>
#include <iostream>
#include <sstream>
#include <string>

int main() {
   const std::string inputString{"Amanda test 123 4.7 A"};
   std::istringstream input{inputString};
   std::string s1;
   std::string s2;
   int i;
   double d;
   char c;

   input >> s1 >> s2 >> i >> d >> c;

   std::cout << "Items extracted from the istringstream object:\n"
      << fmt::format("{}\n{}\n{}\n{}\n{}\n", s1, s2, i, d, c);

   // Пробуем выполнить ввод данных из пустого потока
   if (long value; input >> value) {
      std::cout << fmt::format("\nlong value is: {}\n", value);
   }
   else {
      std::cout << fmt::format("\ninput is empty\n");
   }
}
```

```
Items extracted from the istringstream object:
Amanda
test
123
4.7
A

input is empty
```

Илл. 8.13 | Ввод данных из объекта istringstream

8.18. Необработанные строковые литералы

Напомним, что с символов обратного слеша (\) в строках начинаются escape-последовательности (экранированные последовательности), например \n (перевод строки) и \t (табуляция). Если вы хотите включить в строку символ обратного слеша, вам надо поставить два таких символа подряд, и из-за этого некоторые строки становятся трудными для понимания. Например, в Windows обратный слеш служит для разделения имен папок в пути к файлу. Поэтому в строку, содержащую путь к файлу, приходится включать дополнительные символы:

```
std::string windowsPath{"C:\\MyFolder\\MySubFolder\\MyFile.txt"};
```

Для таких случаев более удобны **необработанные строковые литералы (raw string literals)**, введенные в C++11. Такой литерал может содержать любые символы и записывается в следующем формате:

```
R"(любыеСимволы)"
```

Круглые скобки вокруг символов необработанного строкового литерала обязательны. Компилятор автоматически вставит в такой литерал все необходимые экранирующие символы, то есть преобразует каждый специальный символ (обратный слеш, кавычки и т. д.) в соответствующую escape-последовательность. Используя необработанный строковый литерал, можно записать предыдущую инструкцию так:

```
std::string windowsPath{R"(C:\MyFolder\MySubFolder\MyFile.txt)"};
```

Необработанные строковые литералы могут сделать ваш код более наглядным, особенно если в коде есть регулярные выражения, которые мы обсудим в разделе 8.20. Регулярные выражения часто содержат много символов обратного слеша.

Необработанные строковые литералы могут содержать **разделители (delimiters)** длиной до 16 символов: перед левой (открывающей) круглой скобкой и после правой (закрывающей) круглой скобки:

```
R"MYDELIMITER(.*\d[0-35-9]-\d\d-\d\d)MYDELIMITER"
```

Если в необработанном строковом литерале есть разделители, они должны быть идентичными. Обычно разделители необязательны. Но если последовательность символов такого литерала заканчивается одной или несколькими закрывающими круглыми скобками, то разделители обязательны, иначе компилятор воспримет первую из закрывающих скобок как конец литерала.

Необработанные строковые литералы можно использовать в любых местах исходного кода вместо обычных строковых литералов. В них допустимы даже разрывы строк, которые компилятор заменяет символами перевода строки. Например, необработанный строковый литерал:

```
R"(multiple lines
of text)"
```

будет преобразован компилятором в обычный строковый литерал:

```
"multiple lines\nof text"
```

Внимание: если вы записываете необработанный строковый литерал в несколько строк, в него будут включены все отступы во второй и последующих строках.

8.19. Готовые объекты: чтение и анализ CSV-файла с данными о катастрофе «Титаника»

Файлы в формате CSV (comma-separated values — значения, разделенные запятыми), имеющие расширение `.csv`, предназначены для хранения наборов данных.

Этот формат широко применяется для анализа и обработки «больших данных», а также в сфере искусственного интеллекта, например в системах машинного обучения. В этом разделе мы покажем, как прочесть данные из CSV-файла.

Наборы данных

В интернете опубликовано много бесплатных наборов данных. Например, сайт ресурсов для машинного обучения OpenML содержит более 21 тысячи бесплатных наборов данных в формате CSV:

https://openml.org

Эта страница портала GitHub тоже является отличным источником наборов данных:

https://github.com/awesomedata/awesome-public-datasets

Файл account.csv

Для нашего первого примера мы приготовили простой набор данных и записали его в файл `accounts.csv`, расположенный в папке `ch08`. Этот файл содержит те же данные, которые вы видели на илл. 8.11, но в формате значений, разделенных запятыми:

```
account,name,balance
100,Jones,24.98
200,Doe,345.67
300,White,0.0
400,Stone,-42.16
500,Rich,224.62
```

Первая строка CSV-файла, как правило, содержит имена столбцов. Каждая последующая строка является записью, содержащей значения для этих столбцов. В этом наборе данных есть три столбца, соответствующие переменным `account`, `name` и `balance`.

8.19.1. Чтение CSV-файла с помощью библиотеки rapidcsv

Заголовочная библиотека `rapidcsv`[1] опубликована на портале GitHub:

https://github.com/d99kris/rapidcsv

В ней есть класс `rapidcsv::Document`, который вы можете использовать для чтения и обработки CSV-файлов[2]. Многие другие библиотеки тоже имеют встроенную поддержку формата CSV. Для вашего удобства мы уже поместили библиотеку `rapidcsv` в папку `libraries/rapidcsv`, расположенную в `examples`. Как и в пре-

[1] © 2017, Кристофер Бергрен (Kristofer Berggren). Все права защищены.

[2] Другой популярный формат данных — JavaScript Object Notation (JSON). Существуют библиотеки C++ для чтения и генерации JSON-файлов, например RapidJSON (https://github.com/Tencent/rapidjson) и cereal (https://uscilab.github.io/cereal/index.html; см. раздел 9.22).

дыдущих примерах со сторонними библиотеками, в программу надо включить заголовок, в данном случае <rapidcsv.h> (илл. 8.14, строка 5).

```
 1 // fig08_14.cpp
 2 // Ввод данных из CSV-файла
 3 #include <fmt/format.h>
 4 #include <iostream>
 5 #include <rapidcsv.h>
 6 #include <vector>
 7
 8 int main() {
 9    rapidcsv::Document document{"accounts.csv"}; // Открываем файл по имени
10    std::vector<int> accounts{document.GetColumn<int>("account")};
11    std::vector<std::string> names{
12       document.GetColumn<std::string>("name")};
13    std::vector<double> balances{document.GetColumn<double>("balance")};
14
15    std::cout << fmt::format(
16       "{:<10}{:<13}{:>7}\n", "Account", "Name", "Balance");
17
18    for (size_t i{0}; i < accounts.size(); ++i) {
19       std::cout << fmt::format("{:<10}{:<13}{:>7.2f}\n",
20          accounts.at(i), names.at(i), balances.at(i));
21    }
22 }
```

```
Account   Name     Balance
100       Jones      24.98
200       Doe       345.67
300       White       0.00
400       Stone     -42.16
500       Rich      224.62
```

Илл. 8.14 | Ввод данных из CSV-файла

Для открытия файла `"accounts.csv"`[1] создается и инициализируется объект `rapidcsv::Document` с именем `document` (строка 9). Функции класса `Document` позволяют работать с данными CSV-файла по строкам, по столбцам и по отдельным значениям на пересечениях строк и столбцов. В этом примере поочередно загружаются столбцы, для чего трижды вызывается имеющаяся в шаблоне класса `Document` функция `GetColumn` (строки 10–13). Она возвращает данные указанного столбца в виде вектора `std::vector`, содержащего элементы указанного в угловых скобках типа. Вызов этой функции в строке 10:

```
document.GetColumn<int>("account")
```

возвращает вектор элементов `int`, содержащих значения `account` для каждой записи. Вызовы в строках 11–12 и 13 возвращают векторы элементов `string`

[1] Файл accounts.csv должен находиться в папке с исполняемым файлом программы.

и double соответственно, содержащих значения name и balance. В строках 15–21 полученные данные форматируются и выводятся на экран для подтверждения успешного чтения файла.

Внимание: запятые в полях CSV-файла

Будьте осторожны при работе со строками, содержащими запятые, например "Jones, Sue". Если бы мы случайно сохранили такое имя в двух строках "Jones" и "Sue", то CSV-запись в нашем примере содержала бы четыре поля вместо трех. Программы, работающие с CSV-файлами, обычно рассчитаны на одинаковое количество полей в каждой записи. В противном случае возникают проблемы.

Внимание: пропущенные запятые и лишние запятые в CSV-файлах

Будьте осторожны при создании и обработке CSV-файлов. Например, ваш файл может состоять из записей, в каждой из которых есть четыре значения, разделенных запятыми:

```
100,85,77,9
```

Предположим, вы случайно пропустите одну из запятых:

```
100,8577,9
```

Тогда запись будет содержать только три поля, одно из которых будет содержать неправильное значение 8577.

Предположим, вы случайно поставите две запятые вместо одной:

```
100,85,,77,9
```

Тогда вы получите пять полей вместо четырех, а одно из них будет пустым. Каждая из этих ошибок может привести к неправильной обработке данных программой.

8.19.2. Работа с набором данных «Титаник»

Начинающие специалисты по работе с большими данными часто используют в своей работе набор данных «Титаник»[1]. В нем перечислены все пассажиры и указано, кто из них выжил, когда пароход столкнулся с айсбергом и затонул в своем первом рейсе 15 апреля 1912 года. Давайте введем эти данные в программу и проанализируем.

Набор данных «Титаник» в формате CSV мы поместили в файл titanic.csv, который находится в архиве CPlusPlus20ForProgrammers-master.zip, в каталоге

[1] Набор данных «Titanic» с сайта OpenML.org (https://www.openml.org/d/40945). Авторы: Фрэнк Э. Харрелл-младший (Frank E. Harrell, Jr.) и Томас Кейсон (Thomas Cason). Источник: Vanderbilt Biostatistics (https://biostat.app.vumc.org/wiki/pub/Main/DataSets/titanic.html). В условиях лицензии OpenML (https://www.openml.org/cite) сказано: «Вы можете свободно использовать OpenML и все эмпирические данные и метаданные по лицензии CC-BY» (https://creativecommons.org/licenses/by/4.0).

с примерами кода из главы 8. Вы можете загрузить этот архив на портале GitHub или на сайте Deitel:

https://github.com/pdeitel/CPlusPlus20ForProgrammers

https://www.deitel.com/books/c-plus-plus-20-for-programmers

Ознакомление с данными

Чтобы анализировать данные, надо хорошо понимать, что они собой представляют. Для начала можно просто просмотреть записи в CSV-файле. Если вы откроете файл `titanic.csv` в текстовом редакторе или в Excel, вы увидите, что он состоит из 1309 записей, разделенных запятыми на 14 столбцов (специалисты по анализу данных говорят, что каждый из 1309 объектов характеризуется 14 признаками). Мы будем использовать только четыре столбца:

- `survived`: `1` — пассажир выжил, `0` — не выжил;
- `sex`: `"female"` — женский пол, `"male"` — мужской;
- `age`: возраст; почти все значения — это целые числа, но для некоторых детей в возрасте до года указаны значения с плавающей точкой, поэтому для обработки этого столбца мы выбрали тип `double`;
- `pclass`: `1` — билет первого класса, `2` — второго, `3` — третьего.

Здесь вы можете узнать о значениях всех остальных столбцов:

https://biostat.app.vumc.org/wiki/pub/Main/DataSets/titanic3info.txt

Неизвестные данные

Плохие (то есть недостоверные) и неизвестные значения могут существенно влиять на результаты анализа данных. В наборе данных «Титаник» не указан возраст 263 пассажиров. Отсутствие данных о возрасте обозначено символом `?` в столбце `age` CSV-файла. Для получения статистических данных о возрасте пассажиров мы будем отфильтровывать и игнорировать неизвестные значения. Некоторые аналитики советуют никогда не заменять в неполном наборе данных неизвестные значения «самыми подходящими», а вместо этого передавать такой набор в анализирующую программу, явно обозначив в нем все отсутствующие элементы. Другие аналитики допускают замену, но только при соблюдении строгих правил[1].

---

[1] Эта сноска — часть комментария, отправленного нам 20 июля 2018 года одним из академических рецензентов нашего учебника по Python доктором Элисон Санчес (Alison Sanchez) из школы бизнеса Университета Сан-Диего. Она прокомментировала: «Будьте осторожны, упоминая "замену подходящими значениями" отсутствующих или плохих значений. Строгое предупреждение: не допускается "замена" тех значений, которые повышают статистическую значимость или дают "более обоснованные" или "лучшие" результаты. "Замена" данных не должна превращаться в "подделку" данных. Первое правило, которое должны усвоить студенты, — не удалять и не изменять те значения, которые противоречат их гипотезам. "Подстановка подходящих значений" не означает, что студентам разрешается произвольно изменять значения, чтобы получать желаемые результаты».

Ввод набора данных в программу

В нашей программе, разделенной на части для удобства обсуждения (илл. 8.15), используются некоторые функции библиотеки `ranges` (C++20), рассмотренной в разделе 6.14. Для открытия файла `titanic.csv`[1] создается и инициализируется объект `rapidcsv::Document` с именем `titanic` (строки 15–17). Второй и третий аргументы в строке 16 — это два аргумента по умолчанию, используемые для инициализации объекта `Document` в том случае, когда он создается только по имени CSV-файла. Напомним, что явное указание аргумента функции требует явного указания всех предыдущих аргументов (см. раздел 5.13). Мы указали второй и третий аргументы только для того, чтобы можно было указать четвертый аргумент.

```
 1 // fig08_15.cpp
 2 // Чтение набора данных "Титаник" из CSV-файла с последующим анализом
 3 #include <fmt/format.h>
 4 #include <algorithm>
 5 #include <cmath>
 6 #include <iostream>
 7 #include <numeric>
 8 #include <ranges>
 9 #include <rapidcsv.h>
10 #include <string>
11 #include <vector>
12
13 int main() {
14    // Вводим данные, заменяя отсутствующие данные о возрасте значением NaN
15    rapidcsv::Document titanic{"titanic.csv",
16       rapidcsv::LabelParams{}, rapidcsv::SeparatorParams{},
17       rapidcsv::ConverterParams{true}};
18
```

Илл. 8.15 | Чтение набора данных «Титаник» из CSV-файла с последующим анализом

Аргумент по умолчанию `rapidcsv::LabelParams{}` означает, что первая строка нашего CSV-файла содержит имена столбцов. Аргумент по умолчанию `rapidcsv::SeparatorParams{}` означает, что поля каждой записи разделяются запятыми. Четвертый аргумент:

```
rapidcsv::ConverterParams{true}
```

означает, что `rapidcsv` автоматически заменит в наборе данных все отсутствующие и недопустимые значения. В столбцах с целыми числами каждое такое значение будет преобразовано в `0`, а в столбцах с числами с плавающей точкой — в `NaN` (Not a Number). Это позволит нам загрузить весь столбец `age` в вектор элементов `double`, в том числе обозначенные символом `?` неизвестные значения.

[1] Файл titanic.csv должен находиться в папке с исполняемым файлом программы.

Подготовка данных для анализа

В строках 20–23 функция `GetColumn` класса `rapidcsv::Document` копирует в вектор данные из каждого указанного по имени столбца CSV-файла.

```
19    // GetColumn возвращает данные столбцов в векторах элементов тех же типов
20    auto survived{titanic.GetColumn<int>("survived")};
21    auto sex{titanic.GetColumn<std::string>("sex")};
22    auto age{titanic.GetColumn<double>("age")};
23    auto pclass{titanic.GetColumn<int>("pclass")};
24
```

Выборочный просмотр строк из набора данных «Титаник»

Каждая из 1309 строк содержит данные об одном пассажире. Согласно «Википедии», на борту находилось примерно 1317 пассажиров, 815 из них погибли[1]. Все записи большого набора данных невозможно отобразить на экране одновременно. Для ознакомления с набором данных программа обычно показывает несколько записей с начала и с конца — это позволяет получить о нем общее представление. Инструкции в строках 26–31 отображают первые пять элементов каждого столбца:

```
25    // Выводим на экран первые 5 строк
26    std::cout << fmt::format("First five rows:\n{:<10}{:<8}{:<6}{}\n",
27       "survived", "sex", "age", "class");
28    for (size_t i{0}; i < 5; ++i) {
29       std::cout << fmt::format("{:<10}{:<8}{:<6.1f}{}\n",
30          survived.at(i), sex.at(i), age.at(i), pclass.at(i));
31    }
32
```

```
First five rows:
survived  sex     age   class
1         female  29.0  1
1         male     0.9  1
0         female   2.0  1
0         male    30.0  1
0         female  25.0  1
```

Инструкции в строках 34–40 отображают последние пять элементов каждого столбца. Для этого функция `getRowCount` класса `rapidcsv::Document` возвращает количество строк в наборе данных (строка 36), затем управляющая переменная инициализируется значением, которое на 5 меньше значения `count` (строка 37). Обратите внимание, что неизвестный возраст пассажира обозначен в выходных данных словом `nan`.

[1] «Passengers of the Titanic». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Passengers_of_the_RMS_Titanic.

```
33    // Выводим на экран последние 5 строк
34    std::cout << fmt::format("\nLast five rows:\n{:<10}{:<8}{:<6}{}\n",
35       "survived", "sex", "age", "class");
36    const auto count{titanic.GetRowCount()};
37    for (size_t i{count - 5}; i < count; ++i) {
38       std::cout << fmt::format("{:<10}{:<8}{:<6.1f}{}\n",
39          survived.at(i), sex.at(i), age.at(i), pclass.at(i));
40    }
41
```

```
Last five rows:
survived  sex     age   class
0         female  14.5  3
0         female  nan   3
0         male    26.5  3
0         male    27.0  3
0         male    29.0  3
```

Базовая описательная статистика

Для ознакомления с набором данных аналитики часто используют статистические методы, чтобы описать и обобщить эти данные. Давайте рассчитаем пять базовых статистических характеристик, основываясь на столбце age: количество пассажиров известного возраста, а также средний, минимальный, максимальный и медианный возраст. До начала этих расчетов мы должны очистить данные, удалив все значения nan. Любое выражение, в котором есть nan, дает результат nan. В строках 43–44 мы используем рассмотренный в разделе 6.14 метод фильтрации с помощью библиотеки ranges (C++20), чтобы оставить в векторе age только значения, отличающиеся от nan. Функция isnan (заголовок <cmath>) возвращает true, если значение равно nan. В строках 45–46 мы создаем вектор элементов double с именем cleanAge. Он инициализирует свои элементы путем перебора отфильтрованных результатов в конвейере removeNaN.

```
42    // Удаляем отсутствующие значения возраста с помощью библиотеки ranges
43    auto removeNaN{
44       age | std::views::filter([](const auto& x) {return !isnan(x);})};
45    std::vector<double> cleanAge{
46       std::begin(removeNaN), std::end(removeNaN)};
47
```

Базовая описательная статистика по данным из очищенного столбца age

После очистки данных можно приступить к расчетам. В строке 49 мы сортируем по возрастанию вектор cleanAge, что поможет найти минимальный, максимальный и медианный возраст. А в строке 50 мы получаем количество пассажиров известного возраста: оно равно размеру вектора cleanAge.

```
48    // Описательная статистика для отфильтрованного столбца age
49    std::sort(std::begin(cleanAge), std::end(cleanAge));
50    size_t size{cleanAge.size()};
51    double median{};
52
53    if (size % 2 == 0) { // Медианное значение при четном количестве элементов
54       median = (cleanAge.at(size / 2 - 1) + cleanAge.at(size / 2)) / 2;
55    }
56    else { // Медианное значение при нечетном количестве элементов
57       median = cleanAge.at(size / 2);
58    }
59
60    std::cout << "\nDescriptive statistics for the age column:\n"
61       << fmt::format("Passengers with age data: {}\n", size)
62       << fmt::format("Average age: {:.2f}\n", std::accumulate(
63          std::begin(cleanAge), std::end(cleanAge), 0.0) / size)
64       << fmt::format("Minimum age: {:.2f}\n", cleanAge.front())
65       << fmt::format("Maximum age: {:.2f}\n", cleanAge.back())
66       << fmt::format("Median age: {:.2f}\n", median);
67
```

```
Descriptive statistics for the age column:
Passengers with age data: 1046
Average age: 29.88
Minimum age: 0.17
Maximum age: 80.00
Median age: 28.00
```

В строках 51–58 определяется медианный возраст. Если в векторе `cleanAge` четное количество элементов, медианой является среднее арифметическое двух элементов, расположенных посередине вектора (строка 54); если нечетное — медианой является значение элемента, расположенного посередине (строка 57). В строках 60–66 вся статистика выводится на экран. Средний возраст рассчитывается с помощью алгоритма `accumulate` (строки 62–63): сумма всех значений вектора `cleanAge` делится на его размер. Поскольку вектор уже отсортирован, минимальный и максимальный возраст определяются функциями `front` и `back` класса `vector` (строки 64–65), которые возвращают первый и последний элементы соответственно. Среднее и медианное значения относятся к мерам центральной тенденции. **Мерой центральной тенденции (measure of central tendency)** для набора числовых данных называется такое значение, которое является «центральным» в этом наборе, то есть в каком-то смысле типичным среди остальных.

Итак, мы выяснили, что в наборе данных «Титаник» указан возраст `1046` пассажиров. Их средний возраст — `29.88` лет. Самому младшему из них было чуть больше двух месяцев ($0.17 \cdot 12 = 2.04$), а самому старшему — `80` лет. Медианный возраст пассажиров — `28` лет.

Подсчет количества пассажиров по классам

Теперь выясним, сколько пассажиров было в каждом классе. В строках 69–74 определяется лямбда, которая считает количество пассажиров в указанном классе. Алгоритм `std::ranges::count_if` (C++20) считает все те элементы в своем первом аргументе, для которых лямбда во втором аргументе возвращает `true`. Одно из преимуществ алгоритмов библиотеки `std::ranges` заключается в том, что вам не нужно указывать начало и конец контейнера — эти алгоритмы упрощают код, делая часть работы за вас. Первым аргументом алгоритма `count_if` в этом примере будет вектор элементов `int` с именем `pclass`. Лямбда в строке 72 содержит в списке захвата параметр `classNumber`:

```
[classNumber](int x) {return classNumber == x;}
```

Это называется **захватом (capturing)** параметра: лямбда захватывает `classNumber` из доступной области видимости (из лямбды `countClass`) и использует его в своем теле. По умолчанию захват выполняется по значению, поэтому лямбда в строке 72 получает копию `classNumber`. В главе 14 мы подробнее поговорим о лямбдах с захваченными параметрами и об алгоритмах библиотеки `std::ranges`. В строках 75–77 определяются константы для трех классов пассажиров. В строках 78–80 лямбда `countClass` вызывается для каждого из классов, а в строках 82–84 результаты выводятся на экран.

```
68    // Количество пассажиров первого, второго и третьего класса
69    auto countClass{
70       [](const auto& column, const int classNumber) {
71          return std::ranges::count_if(column,
72             [classNumber](int x) {return classNumber == x;});
73       }
74    };
75    constexpr int firstClass{1};
76    constexpr int secondClass{2};
77    constexpr int thirdClass{3};
78    const auto firstCount{countClass(pclass, firstClass)};
79    const auto secondCount{countClass(pclass, secondClass)};
80    const auto thirdCount{countClass(pclass, thirdClass)};
81
82    std::cout << "\nPassenger counts by class:\n"
83       << fmt::format("1st: {}\n2nd: {}\n3rd: {}\n\n",
84          firstCount, secondCount, thirdCount);
85
```

```
Passenger counts by class:
1st: 323
2nd: 277
3rd: 709
```

Подсчет количества выживших пассажиров

В строках 87–89 определяется лямбда, которая считает количество выживших пассажиров с помощью алгоритма `std::ranges::count_if` (C++20). Напомним, что столбец `survived` содержит `1` или `0` для обозначения выживших или погибших соответственно. Такие значения C++ может обрабатывать как `true` (`1`) или `false` (`0`), поэтому лямбда в строке 88 просто возвращает значение столбца:

```
[](auto x) {return x;}
```

Если это значение равно 1, алгоритм `count_if` добавляет элемент к сумме. Чтобы определить количество погибших, в строке 92 из размера вектора `survived` вычитается `survivorCount`. В строке 94 вычисляется процент выживших.

```
86    // Подсчет количества выживших пассажиров
87    const auto survivorCount{
88       std::ranges::count_if(survived, [](auto x) {return x;})
89    };
90
91    std::cout << fmt::format("Survived count: {}\nDied count: {}\n",
92       survivorCount, survived.size() - survivorCount);
93    std::cout << fmt::format("Percent who survived: {:.2f}%\n\n",
94       100.0 * survivorCount / survived.size());
95
```

```
Survived count: 500
Died count: 809
Percent who survived: 38.20%
```

Подсчет количества выживших пассажиров по категориям

В строках 97–117 перебираются все элементы столбца `survived`, их значения `1` и `0` используются в качестве условия (строка 104). Для каждого выжившего увеличиваются счетчики, соответствующие его полу (`survivingWomen` и `survivingMen` в строке 105) и классу (`surviving1st`, `surviving2st`, `surviving3st` в строках 107–115). Результаты будут использованы для вычисления процентов.

```
96    // Количество выживших пассажиров по категориям
97    int survivingMen{0};
98    int survivingWomen{0};
99    int surviving1st{0};
100   int surviving2nd{0};
101   int surviving3rd{0};
102
103   for (size_t i{0}; i < survived.size(); ++i) {
104      if (survived.at(i)) {
105         sex.at(i) == "female" ? ++survivingWomen : ++survivingMen;
106
107         if (firstClass == pclass.at(i)) {
108            ++surviving1st;
109         }
```

```
110         else if (secondClass == pclass.at(i)) {
111            ++surviving2nd;
112         }
113         else { // Третий класс
114            ++surviving3rd;
115         }
116      }
117   }
118
```

Вычисление процентов

В строках 120–129 вычисляются и отображаются проценты:

- выживших женщин;
- выживших мужчин;
- выживших пассажиров первого класса;
- выживших пассажиров второго класса;
- выживших пассажиров третьего класса.

Итак, среди выживших было примерно две трети женщин, а наибольший шанс выжить был у пассажиров первого класса.

```
119   // Процент выживших пассажиров по категориям
120   std::cout << fmt::format("Female survivor percentage: {:.2f}%\n",
121         100.0 * survivingWomen / survivorCount)
122      << fmt::format("Male survivor percentage: {:.2f}%\n\n",
123         100.0 * survivingMen / survivorCount)
124      << fmt::format("1st class survivor percentage: {:.2f}%\n",
125         100.0 * surviving1st / survivorCount)
126      << fmt::format("2nd class survivor percentage: {:.2f}%\n",
127         100.0 * surviving2nd / survivorCount)
128      << fmt::format("3rd class survivor percentage: {:.2f}%\n",
129         100.0 * surviving3rd / survivorCount);
130 }
```

```
Female survivor percentage: 67.80%
Male survivor percentage: 32.20%

1st class survivor percentage: 40.00%
2nd class survivor percentage: 23.80%
3rd class survivor percentage: 36.20%
```

8.20. Готовые объекты: основы регулярных выражений

Иногда вам бывает нужно найти в тексте все фрагменты, соответствующие определенным шаблонам: телефонные номера, адреса электронной почты, почтовые

индексы, адреса веб-страниц и т. д. **Регулярное выражение (regular expression**, или сокращенно **regex**) — это шаблон, который может сравниваться на совпадение с проверяемыми строками и их фрагментами (подстроками). Регулярные выражения помогают быстро извлекать нужные данные из неструктурированного текста, например из сообщений в соцсетях. Кроме того, регулярные выражения значительно упрощают проверку формата, необходимую перед обработкой данных.

Проверка формата данных

Как правило, любую обработку введенных пользователем текстовых данных надо начинать с проверки их формата регулярными выражениями. Например, вы можете проверить, что:

- почтовый индекс США состоит из пяти цифр (например, `02215`) или из девяти цифр, разделенных дефисом после пятой цифры (например, `02215-4775`);
- имя пользователя содержит только буквы, пробелы, апострофы и дефисы;
- адрес электронной почты содержит только допустимые символы в допустимой последовательности;
- номер социального страхования США содержит три цифры, дефис, две цифры, дефис и еще четыре цифры, а также соответствует правилам о конкретных числах, имеющихся в каждой группе цифр.

Иногда вам придется составлять ваши собственные регулярные выражения, необходимые для всевозможных задач проверки данных. Вам помогут эти веб-сайты:

https://regex101.com

https://regexr.com

http://www.regexlib.com

https://www.regular-expressions.info

На этих и других сайтах есть бесплатные репозитории готовых регулярных выражений, которые можно копировать и использовать. Кроме того, многие сайты предлагают тестирование регулярных выражений на соответствие вашим требованиям.

Другие применения регулярных выражений

Регулярные выражения удобны для следующих задач:

- **скрейпинг (scraping)**, то есть извлечение данных из текста, например копирование всех URL-адресов с веб-страницы в файл;
- очистка данных: удаление ненужных данных и дубликатов, обработка неполных данных, исправление опечаток, приведение данных к единому формату, удаление ненужного форматирования, изменение регистра букв и многие другие операции со строками;
- преобразование данных в другие форматы, например замена всех нестандартных разделителей в наборе данных (пробелов или символов табуляции) на

запятые для последующей обработки в формате CSV. (Мы рассматривали CSV-файлы в разделе 8.19.)

Диалекты регулярных выражений в C++

C++ поддерживает несколько вариантов грамматики регулярных выражений, часто называемых **диалектами (flavors)**. По умолчанию в C++ используется слегка измененная версия регулярных выражений из языка ECMAScript[1]. Полный список поддерживаемых грамматических вариантов (диалектов) приведен здесь:

https://en.cppreference.com/w/cpp/regex/syntax_option_type

8.20.1. Сравнение целых строк с шаблонами

Чтобы использовать регулярные выражения, надо включить в программу заголовок `<regex>`, предоставляющий классы и функции для их поддержки[2]. Наша следующая программа, разделенная на части для удобства обсуждения (илл. 8.16), демонстрирует операции, в которых целая строка текста сравнивается на совпадение с тем или иным шаблоном (регулярным выражением).

Сравнение литералов

Функция `regex_match` из заголовка `<regex>` возвращает `true`, если вся строка в первом аргументе совпадает с шаблоном во втором аргументе. По умолчанию сравнение с шаблоном чувствительно к регистру; далее мы покажем, как выполнять аналогичные операции без учета регистра. Давайте для начала сравним между собой короткие последовательности символов. Мы создаем объект `regex` с именем `r1` (строка 9), содержащий простой шаблон `"02215"` из пяти цифровых литералов. При сравнении этого шаблона с другой строкой совпадение будет найдено только в том случае, если сравниваемая строка ничем не отличается от шаблона, то есть состоит из этих же пяти литералов в этой же последовательности. Мы сравниваем шаблон со строками `"02215"` и `"51220"`, дважды вызывая функцию `regex_match` (строка 12). Обе сравниваемые строки составлены из одних и тех же цифровых литералов, но только в строке `"02215"` последовательность цифр такая же, как в шаблоне.

```
1 // fig08_16.cpp
2 // Сравнение целых строк с шаблонами
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <regex>
6
7 int main() {
8    // Шаблон: фиксированная последовательность литералов          Окончание ⇨
```

[1] «Modified ECMAScript Regular Expression Grammar». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/regex/ecmascript.

[2] Некоторые программисты C++ предпочитают сторонние библиотеки регулярных выражений, например RE2 или PCRE. Список таких библиотек есть на портале GitHub: https://github.com/fffaraz/awesome-cpp#regular-expression.

```
 9    std::regex r1{"02215"};
10    std::cout << "Matching against: 02215\n"
11       << fmt::format("02215: {}; 51220: {}\n\n",
12          std::regex_match("02215", r1), std::regex_match("51220", r1));
13
```

```
Matching against: 02215
02215: true; 51220: false
```

Илл. 8.16 | Сравнение целых строк с шаблонами

Метасимволы и символьные классы

Как правило, регулярные выражения содержат специальные символы, называемые **метасимволами (metacharacters)**:

```
[] {} () \ * + ^ $ ? . |
```

С метасимвола \ начинаются все **символьные классы (character classes)**. В таблице перечислены важнейшие символьные классы и указано, каким символам они соответствуют.

Символьный класс	Соответствие
`\d`	Любая цифра (0–9)
`\D`	Любой символ, кроме цифры
`\s`	Любой символ пропуска (пробел, табуляция, перевод строки и др.)
`\S`	Любой символ, кроме символа пропуска
`\w`	Любой словообразующий символ (word character), которым может быть любая заглавная или строчная буква английского алфавита, любая цифра или символ подчеркивания
`\W`	Любой символ, кроме словообразующего символа

Чтобы включить в шаблон любой метасимвол в виде обычного литерала, добавьте перед ним обратный слеш. Например, шаблон `\$` соответствует символу доллара (`$`), а шаблон `\\` соответствует обратному слешу (\).

Шаблон с квантификатором: пять цифр подряд

В следующей части программы мы моделируем проверку почтового индекса, который должен состоять из пяти цифр. В регулярном выражении `\d{5}`, которое мы записали в виде необработанного строкового литерала (строка 15), `\d` — это символьный класс, соответствующий любой цифре (`0-9`). Символьный класс — это escape-последовательность, соответствующая какому-либо одному символу. Чтобы найти пять любых цифр подряд, мы добавили **квантификатор (quantifier)** `{5}`, который повторяет `\d` пять раз, что аналогично шаблону `\d\d\d\d\d`. Второй вызов функции `regex_match` (строка 18) возвращает `false`, поскольку почтовый индекс `"9876"` некорректен: в нем только четыре цифры.

```
14    // Шаблон: любая строка из 5 цифр
15    std::regex r2{R"(\d{5})"};
16    std::cout << R"(Matching against: \d{5})" << "\n"
17       << fmt::format("02215: {}; 9876: {}\n\n",
18          std::regex_match("02215", r2), std::regex_match("9876", r2));
19
```

```
Matching against: \d{5}
02215: true; 9876: false
```

Пользовательские символьные классы

Символами в квадратных скобках [] определяется **пользовательский символьный класс (custom character class)**, который тоже соответствует какому-то одному символу. Например, `[aeiou]` — это любая из пяти гласных букв английского алфавита в нижнем регистре; `[A-Z]` — любая заглавная буква от `A` до `Z`; `[a-z]` — любая строчная буква от `a` до `z`; `[a-zA-Z]` — любая буква английского алфавита. В строке 21 мы определяем пользовательский символьный класс для проверки имени пользователя, которое не должно содержать пробелы и знаки препинания.

```
20    // Шаблон: любое слово, которое начинается с заглавной буквы A-Z
21    std::regex r3{"[A-Z][a-z]*"};
22    std::cout << "Matching against: [A-Z][a-z]*\n"
23       << fmt::format("Wally: {}; eva: {}\n\n",
24          std::regex_match("Wally", r3), std::regex_match("eva", r3));
25
```

```
Matching against: [A-Z][a-z]*
Wally: true; eva: false
```

В имени человека может быть много букв. В регулярном выражении `r3` (строка 21) класс `[A-Z]` соответствует одной заглавной букве, а класс `[a-z]*` — любому (в том числе нулевому) количеству строчных букв. Квантификатор `*` после какого-либо элемента регулярного выражения обозначает, что этот элемент (в данном случае класс `[a-z]`, то есть любая буква от `a` до `z`) повторяется ноль или более раз подряд. Например, шаблон `[A-Z][a-z]*` соответствует строке `"Amanda"`, `"Bo"` и даже `"E"`.

Пользовательский символьный класс, который начинается с `^`, соответствует любому символу, кроме указанных после `^`. Например, шаблон `[^a-z]` (строка 27) обозначает любой символ, кроме строчной буквы в диапазоне от `a` до `z`.

```
26    // Шаблон: любой символ, кроме строчной буквы a-z
27    std::regex r4{"[^a-z]"};
28    std::cout << "Matching against: [^a-z]\n"
29       << fmt::format("A: {}; a: {}\n\n",
30          std::regex_match("A", r4), std::regex_match("a", r4));
31
```

```
Matching against: [^a-z]
A: true; a: false
```

Метасимволы в пользовательском классе обрабатываются как литералы, то есть как обычные символы. Например, шаблон **[*+$]** (строка 33) соответствует любому из трех символов: *, + или $.

```
32    // Метасимволы в пользовательском классе символов считаются литералами
33    std::regex r5{"[*+$]"};
34    std::cout << "Matching against: [*+$]\n"
35       << fmt::format("*: {}; !: {}\n\n",
36          std::regex_match("*", r5), std::regex_match("!", r5));
37
```

```
Matching against: [*+$]
*: true; !: false
```

Сравнение квантификаторов + и *

Если в имени пользователя должна быть хотя бы одна строчная буква, замените квантификатор * (строка 21) на + (строка 39). Квантификатор + после какого-либо элемента регулярного выражения обозначает, что этот элемент повторяется один или более раз подряд.

```
38    // Любое слово с первой заглавной буквой A-Z и второй строчной буквой a-z
39    std::regex r6{"[A-Z][a-z]+"};
40    std::cout << "Matching against: [A-Z][a-z]+\n"
41       << fmt::format("Wally: {}; E: {}\n\n",
42       std::regex_match("Wally", r6), std::regex_match("E", r6));
43
```

```
Matching against: [A-Z][a-z]+
Wally: true; E: false
```

Квантификаторы * и + являются **жадными (greedy)**. Жадным называется квантификатор, обозначающий наибольшее возможное количество последовательных символов. Например, регулярное выражение `[A-Z][a-z]+` соответствует словам `"Al"`, `"Eva"`, `"Samantha"`, `"Benjamin"` и любым другим словам, начинающимся с заглавной буквы, за которой следует хотя бы одна строчная буква. Вы можете добавить символ ? после квантификатора * и +, чтобы сделать его **ленивым (lazy)**: *? или +?. Ленивым называется квантификатор, обозначающий наименьшее возможное количество последовательных символов.

Другие квантификаторы

Квантификатор ? сам по себе обозначает, что элемент слева от него повторяется ноль или один раз. В регулярном выражении `labell?ed` (строка 45) таким элементом является литерал `"l"`. Поэтому в двух первых вызовах функции `regex_match` (строки 48–49) шаблон соответствует слову `labelled` (так пишут в Великобритании) и `labeled` (так пишут в США). Но в третьем вызове `regex_match` (строка 50) шаблон не соответствует слову `labellled`, которое написано с ошибкой.

```
44    // Элемент входит в шаблон не более одного раза
45    std::regex r7{"labell?ed"};
46    std::cout << "Matching against: labell?ed\n"
47       << fmt::format("labelled: {}; labeled: {}; labellled: {}\n\n",
48          std::regex_match("labelled", r7),
49          std::regex_match("labeled", r7),
50          std::regex_match("labellled", r7));
51
```

```
Matching against: labell?ed
labelled: true; labeled: true; labellled: false
```

Вы можете использовать квантификатор `{n,}`, чтобы обозначить `n` или более последовательных повторений элемента слева от него. Регулярное выражение в строке 53 соответствует строкам, состоящим как минимум из трех цифр подряд.

```
52    // Элемент входит в шаблон не менее трех раз подряд
53    std::regex r8{R"(\d{3,})"};
54    std::cout << R"(Matching against: \d{3,})" << "\n"
55       << fmt::format("123: {}; 1234567890: {}; 12: {}\n\n",
56          std::regex_match("123", r8),
57          std::regex_match("1234567890", r8),
58          std::regex_match("12", r8));
59
```

```
Matching against: \d{3,}
123: true; 1234567890: true; 12: false
```

Вы можете использовать квантификатор `{n,m}`, чтобы обозначить от `n` до `m` (включительно) последовательных повторений элемента слева от него. Регулярное выражение в строке 61 соответствует строке, в которой от трех до шести цифр.

```
60    // Элемент входит в шаблон не менее трех и не более шести раз подряд
61    std::regex r9{R"(\d{3,6})"};
62    std::cout << R"(Matching against: \d{3,6})" << "\n"
63       << fmt::format("123: {}; 123456: {}; 1234567: {}; 12: {}\n",
64          std::regex_match("123", r9), std::regex_match("123456", r9),
65          std::regex_match("1234567", r9), std::regex_match("12", r9));
66 }
```

```
Matching against: \d{3,6}
123: true; 123456: true; 1234567: false; 12: false
```

8.20.2. Замена подстрок

Заголовок `<regex>` предоставляет функцию `regex_replace` для замены фрагментов в строке. В следующем примере мы преобразуем строку элементов, разделенных табуляцией, в строку элементов, разделенных запятыми (илл. 8.17). Функция `regex_replace` (строка 13) получает три аргумента:

- строку для поиска (`"1\t2\t3\t4"`);
- регулярное выражение (символ табуляции `"\t"`);
- текст для замены (`","`).

Функция возвращает измененную строку. Инструкция

```
std::regex{"\t"}
```

создает временный объект `regex`, инициализирует его и сразу передает в функцию `regex_replace`. Это удобно, если регулярное выражение используется только один раз.

```
// fig08_17.cpp
// Замена фрагментов в строке
#include <fmt/format.h>
#include <iostream>
#include <regex>
#include <string>

int main() {
   // Заменяем символы табуляции запятыми
   std::string s1{"1\t2\t3\t4"};
   std::cout << fmt::format("Original string: {}\n", R"(1\t2\t3\t4)")
      << fmt::format("After replacing tabs with commas: {}\n",
         std::regex_replace(s1, std::regex{"\t"}, ","));
}
```

```
Original string: 1\t2\t3\t4
After replacing tabs with commas: 1,2,3,4
```

Илл. 8.17 | Замена фрагментов в строке

8.20.3. Поиск в тексте по шаблонам

Вы можете искать в тексте интересующие вас фрагменты с помощью функции `regex_search` (илл. 8.18), которая возвращает `true`, если какой-либо фрагмент строки соответствует шаблону. Эта функция также может предоставить вам доступ к найденному фрагменту через объект шаблона класса `match_results`, который вы передаете в качестве аргумента. Существуют псевдонимы `match_results` для разных типов строк:

- для поиска в `std::strings` используйте `smatch` (произносится как «эс мач»);
- для поиска в строках в стиле C и в строковых литералах используйте `cmatch` (произносится как «си мач»).

На данный момент заголовок `<regex>` не содержит средств для работы с представлениями `string_view`.

Поиск фрагмента без уточнения его места в строке

Каждый из вызовов функции `regex_search` в строках 14–16 выполняет поиск в строке `s1` (`"Programming is fun"`) первого с начала строки фрагмента, соответствующего регулярному выражению. Каждое из трех регулярных выражений — это простая последовательность символов: `"Programming"`, `"fun"` и `"fn"`. Версия функции `regex_search` с двумя аргументами возвращает `true` или `false` в зависимости от того, есть ли в строке искомый фрагмент.

```
1 // fig08_18.cpp
2 // Поиск в тексте по шаблонам
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <regex>
6 #include <string>
7
8 int main() {
9    // Простой поиск: есть ли в строке указанные фрагменты?
10   std::string s1{"Programming is fun"};
11   std::cout << fmt::format("s1: {}\n\n", s1);
12   std::cout << "Search anywhere in s1:\n"
13      << fmt::format("Programming: {}; fun: {}; fn: {}\n\n",
14         std::regex_search(s1, std::regex{"Programming"}),
15         std::regex_search(s1, std::regex{"fun"}),
16         std::regex_search(s1, std::regex{"fn"}));
17
```

```
s1: Programming is fun
Search anywhere in s1:
Programming: true; fun: true; fn: false
```

Илл. 8.18 | Поиск в тексте по шаблонам

Поиск фрагмента без учета регистра букв

Константы `regex_constants` из заголовка `<regex>` позволяют настраивать параметры поиска. Например, по умолчанию поиск чувствителен к регистру, но константа `regex_constants::icase` позволяет искать соответствия без учета регистра.

```
18   // Игнорируем регистр букв
19   std::string s2{"SAM WHITE"};
20   std::smatch match; // Для сохранения фрагмента, соответствующего шаблону
21   std::cout << fmt::format("s2: {}\n\n", s2);
22   std::cout << "Case insensitive search for Sam in s2:\n"
23      << fmt::format("Sam: {}\n", std::regex_search(s2, match,
24         std::regex{"Sam", std::regex_constants::icase}))
25      << fmt::format("Matched text: {}\n\n", match.str());
26
```

```
s2: SAM WHITE
Case insensitive search for Sam in s2:
Sam: true
Matched text: SAM
```

В строках 23–24 вызывается версия функции `regex_search` с тремя аргументами:

- строка для поиска (`s2`);
- объект `smatch` (`match`), в котором будет сохранен соответствующий фрагмент, если он будет найден;
- регулярное выражение (строка 24).

Здесь мы создали регулярное выражение со вторым аргументом:

```
std::regex{"Sam", std::regex_constants::icase}
```

Оно предназначено для поиска слова `"Sam"` независимо от регистра букв. Фрагмент `"SAM"` в строке `s2` соответствует регулярному выражению `"Sam"`, поскольку состоит из тех же букв, только заглавных. Для отображения соответствующего фрагмента в строке 25 вызывается функция `str` объекта `match`, возвращающая его строковое представление.

Поиск всех соответствующих фрагментов по всему тексту

Теперь мы выделим из текста все телефонные номера следующей формы: `###-###-####`. Этот код находит в строке `contact` каждый фрагмент, соответствующий регулярному выражению `phone` (строка 30), и выводит его на экран.

```
27    // Поиск всех соответствующих фрагментов
28    std::string contact{
29       "Wally White, Home: 555-555-1234, Work: 555-555-4321"};
30    std::regex phone{R"(\d{3}-\d{3}-\d{4})"};
31
32    std::cout << fmt::format("Finding phone numbers in:\n{}\n", contact);
33    while (std::regex_search(contact, match, phone)) {
34       std::cout << fmt::format(" {}\n", match.str());
35       contact = match.suffix();
36    }
37 }
```

```
Finding phone numbers in:
Wally White, Home: 555-555-1234, Work: 555-555-4321
555-555-1234
555-555-4321
```

Цикл `while` выполняется, пока `regex_search` возвращает `true`, то есть пока функция находит соответствия. Каждая итерация цикла:

- отображает фрагмент, соответствующий регулярному выражению (строка 34);

- заменяет строку `contact` результатом вызова функции `suffix` объекта `match` (строка 35), возвращающей ту часть строки, в которой еще не производился поиск. Эта новая строка `contact` используется в следующем вызове функции `regex_search`.

8.21. Итоги

В этой главе мы рассмотрели целый ряд операций со строками класса `std::string`: присваивание, конкатенацию, сравнение, поиск фрагментов и обмен данными между строками. Мы также рассказали о функциях класса `std::string`, предназначенных для определения характеристик строки; для поиска, замены и вставки символов в строку; для взаимных преобразований между строками `std::string` и строками в стиле C; для взаимных преобразований между строками и числами.

Мы продемонстрировали создание и обработку текстовых файлов с помощью функций из заголовка `<fstream>` и файловых потоков ввода-вывода. Затем мы перешли к обработке строк с помощью строковых потоков ввода-вывода. В первом примере применения готовых объектов мы прочитали набор данных «Титаник» из CSV-файла с помощью библиотеки с открытым исходным кодом, а затем проанализировали эти данные. Во втором примере мы использовали регулярные выражения для поиска и замены фрагментов текста.

Итак, к концу главы 8 вы изучили общие принципы работы с управляющими инструкциями, функциями, массивами, векторами, строками и файлами. В наших примерах с готовыми объектами вы видели, что в программах на C++ часто создаются и используются объекты, выполняющие много полезной работы. В главе 9 вы узнаете, как создавать свои собственные классы и использовать объекты этих классов в программах. Мы начнем рассматривать внутреннее устройство классов и связанные с ним концепции разработки программного обеспечения.



Пользовательские классы

В этой главе вы:

- определите пользовательский класс и создадите объекты этого класса;
- реализуете поведение класса в виде функций, а атрибуты класса — в виде элементов данных;
- научитесь обращаться к скрытым элементам данных класса через его открытые функции, сохраняя инкапсуляцию данных;
- инициализируете конструктором класса элементы данных создаваемых объектов;
- отделите интерфейс класса от реализации класса для повторного использования;
- научитесь обращаться к элементам класса с помощью операторов «точка» (.) и «стрелка» (->);
- узнаете, как деструкторы выполняют завершающую обработку перед уничтожением объектов;
- научитесь присваивать элементы данных одного объекта элементам данных другого объекта;
- научитесь создавать объекты, содержащие другие объекты;
- познакомитесь с дружественными функциями и классами;
- научитесь обращаться к нестатическим элементам класса через указатель `this`;
- научитесь работать со статическими элементами данных и функциями класса;
- создадите агрегатные типы данных и инициализируете их элементы назначенными инициализаторами;
- примените готовые объекты библиотеки `cereal` для сериализации объектов в формате JSON.

9.1. Введение[1]

В разделе 1.4 мы популярно рассказали об основах объектно-ориентированного программирования, рассмотрев классы, объекты, элементы данных класса (атрибуты объектов) и функции класса (определяющие поведение объектов). Затем в учебных примерах с готовыми объектами вы создавали объекты классов, написанных другими программистами, и вызывали функции этих классов, тем самым поручая объектам выполнение сложных задач. Для этого вам даже не надо было знать, что происходит внутри этих классов в процессе работы.

В этой главе мы приступим к углубленному изучению объектно-ориентированного программирования: теперь вы будете создавать **пользовательские классы (custom classes)**, то есть свои собственные. C++ является **расширяемым языком программирования (extensible programming language)**. Это означает, что каждый созданный вами класс становится новым типом данных, который вы можете использовать для создания объектов. Некоторые команды профессиональных разработчиков в ИТ-компаниях пишут приложения, содержащие сотни и даже тысячи пользовательских классов.

9.2. Тест-драйв объекта пользовательского класса Account

Мы начнем с трех примеров кода, в которых создаются объекты класса `Account`, моделирующего простейший банковский счет. Для начала рассмотрим функцию `main` и выходные данные программы, чтобы вы увидели объект нашего первого класса `Account` в действии. Чтобы вам было проще подготовиться к работе над большими программами, которые вы встретите в последующих главах и в ИТ-отрасли, мы определили функцию `main` и класс `Account` в отдельных файлах, как это принято в ИТ-отрасли: функцию `main` — в основном файле программы `AccountTest.cpp` (илл. 9.1), а класс `Account` — в заголовочном файле `Account.h` (илл. 9.2).

```
1 // Fig. 9.1: AccountTest.cpp
2 // Создание и применение объекта класса Account
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <string>
6 #include "Account.h"
7
8 int main() {
9    Account myAccount{}; // Создаем объект myAccount класса Account
10
11    // Отображаем начальное имя клиента в объекте myAccount (пустую строку)
12    std::cout << fmt::format("Initial account name: {}\n",
13       myAccount.getName());
14
```

Окончание ⇨

[1] Эта глава развивает терминологию и концепции, кратко представленные в разделе 1.4.

```
15    // Запрашиваем и получаем имя клиента
16    std::cout << "Enter the account name: ";
17    std::string name{};
18    std::getline(std::cin, name); // Читаем строку текста с клавиатуры
19    myAccount.setName(name); // Сохраняем введенное имя в объекте myAccount
20
21    // Отображаем имя клиента, сохраненное в объекте myAccount
22    std::cout << fmt::format("Updated account name: {}\n",
23       myAccount.getName());
24 }
```

```
Initial account name:
Enter the account name: Jane Green
Updated account name: Jane Green
```

Илл. 9.1 | Создание и применение объекта класса Account

Создание объекта — экземпляра класса

Как правило, вы не сможете вызывать функции класса, пока не создадите объект этого класса[1]. В строке 9 создается объект класса `Account` с именем `myAccount`:

```
Account myAccount{}; // Создаем объект myAccount класса Account
```

Иными словами, создается переменная `myAccount` типа `Account` — это тип класса, определенного в файле `Account.h` (илл. 9.2).

Заголовок пользовательского класса

Когда мы объявляем переменную типа `int`, компилятор заранее знает, что `int` — это один из фундаментальных типов, встроенных в язык C++. Но когда в строке 9 мы объявляем переменную типа `Account`, компилятор заранее не знает, что такое `Account`, потому что это **пользовательский тип (user-defined type)**.

Правильно написанные и подготовленные пользовательские классы могут повторно использоваться другими программистами. Определение класса, предназначенного для повторного использования, обычно помещают в **заголовок (header)** — файл с расширением `.h`[2]. Затем, если вам нужна функциональность этого класса, вы просто включаете в свою программу этот заголовок. Именно так вы получали в свое распоряжение классы стандартной библиотеки C++ и сторонних библиотек в примерах кода из предыдущих глав книги.

В строке 6 мы сообщаем компилятору, что такое `Account`, включив в программу соответствующий заголовок:

```
#include "Account.h"
```

Если бы мы этого не сделали, компилятор выдавал бы сообщения об ошибках при каждом упоминании «незнакомого» класса `Account` или каких-либо его эле-

---

[1] В разделе 9.20 вы увидите, что есть исключение из этого правила: статические функции.

[2] Заголовки стандартной библиотеки C++, такие как <iostream>, не используют расширение .h, а некоторые программисты предпочитают расширение .hpp.

ментов. Имя заголовка, который вы определили в своей программе, помещается в двойные кавычки (""), а не в угловые скобки (<>). Двойные кавычки сообщают компилятору, что заголовок с этим именем надо сначала искать в папке с основным файлом программы `AccountTest.cpp` (илл. 9.1), а только потом — в папке для заголовков по умолчанию[1].

Вызов функции getName класса Account

Функция `getName` класса `Account` возвращает имя клиента банка (значение переменной `m_name`), хранящееся в конкретном объекте класса `Account`. В строке 13 мы вызываем функцию `getName` для только что созданного объекта `MyAccount`, чтобы получить исходное имя клиента, которое является пустой строкой. Вскоре мы расскажем об этом подробнее.

Вызов функции setName класса Account

Функция `setName` класса `Account` сохраняет имя клиента банка в конкретном объекте класса `Account`. В строке 19 мы вызываем функцию `setName` для объекта `myAccount`, чтобы сохранить в этом объекте значение переменной `name`.

Отображение имени клиента, введенного пользователем

В строке 23 функция `getName` снова вызывается для подтверждения, что объект `myAccount` содержит введенное пользователем имя клиента.

9.3. Класс Account с элементом данных, set-функцией и get-функцией

Теперь, когда вы увидели, как работает класс `Account` (илл. 9.1), рассмотрим его внутреннее устройство.

9.3.1. Определение класса

Класс `Account` (илл. 9.2) содержит элемент данных `m_name` (строка 19) — переменную для хранения имен клиентов банка. Каждый из объектов класса имеет собственную копию элементов данных этого класса[2]. В разделе 9.6 мы добавим в класс `Account` элемент данных `balance` — переменную для хранения «денег» клиентов. На данный момент класс `Account` содержит две функции:

- `setName` (строки 10–12) — для записи имени клиента;
- `getName` (строки 15–17) — для чтения имени клиента.

---

[1] Ваши пользовательские заголовки классов можно включать в программу с помощью угловых скобок, если вы включите папку вашей программы в путь, указанный компилятору для поиска заголовков по умолчанию. (Вы уже включали таким образом сторонние библиотеки в некоторых примерах с готовыми объектами.)

[2] За исключением статических элементов данных, которые являются общими для всех объектов класса. Вы узнаете о них в разделе 9.20.

```
1 // Fig. 9.2: Account.h
2 // Класс Account с элементом данных
3 // и функциями для его записи (set) и чтения (get)
4 #include <string>
5 #include <string_view>
6
7 class Account {
8 public:
9    // Set-функция класса, записывающая в объект значение элемента данных
10   void setName(std::string_view name) {
11      m_name = name; // Записываем в m_name имя клиента
12   }
13
14   // Get-функция класса, читающая из объекта значение элемента данных
15   const std::string& getName() const {
16      return m_name; // Возвращаем имя клиента в точку вызова функции
17   }
18 private:
19   std::string m_name; // Элемент данных, хранящий имя клиента банка
20 }; // Конец класса Account
```

Илл. 9.2 | Класс Account с элементом данных и функциями для его записи (set) и чтения (get)

Ключевое слово class и тело класса

Определение класса начинается с ключевого слова `class` (строка 7), за которым следует имя класса (`Account`). Когда вы пишете определение класса, соблюдайте общепринятое соглашение:

- каждое слово в имени класса начинается с заглавной буквы;
- имена **элементов данных (data members)** класса и **функций класса (member functions)**, также называемых функциями-членами или функциями-элементами, начинаются со строчной буквы, а каждое последующее слово в таком имени начинается с заглавной буквы.

Тело каждого класса заключено в фигурные скобки (строки 7 и 20). Определение класса обязательно завершается точкой с запятой (строка 20).

Элемент данных m_name типа std::string

Напомним из раздела 1.4, что у объектов есть атрибуты. Они реализованы как элементы данных. Каждый объект содержит собственные копии этих данных на протяжении своего жизненного цикла, то есть до тех пор, пока он существует в памяти компьютера. Обычно класс содержит функции, которые оперируют данными конкретных объектов этого класса.

Элементы данных объявляются внутри определения класса, но вне функций класса. В строке 19 объявляется элемент данных с именем `m_name`:

```
std::string m_name; // Элемент данных, хранящий имя клиента банка
```

Префикс `m_` нужен по общепринятому соглашению об именовании, он указывает, что эта переменная — элемент данных. Если вы создадите много объектов класса `Account`, каждый из них будет иметь свою собственную переменную `m_name`.

Поскольку `m_name` является элементом данных, с ним можно выполнять разные действия, вызывая функции класса. Напомним, что переменные типа `string` по умолчанию инициализируются пустой строкой (`""`), поэтому первый вызов функции `getName` из `main` (илл. 9.1, строка 13) не возвращает никакого имени клиента.

По общепринятому соглашению, программисты C++ обычно размещают элементы данных класса в конце его тела. Вы можете объявлять элементы данных в любых местах за пределами определений функций, но разброс элементов данных по всему классу делает код трудным для понимания.

Квалификатор std:: для стандартных библиотечных компонентов в заголовках

В заголовке `Account.h` (илл. 9.2) мы используем квалификатор `std::` при обращении к `string_view` (строка 10) и к `string` (строки 15 и 19). В заголовках не должно быть ни директив `using`, ни объявлений `using`, указывающих на глобальную область видимости. Это необходимо во избежание конфликтов имен, так как заголовки включаются в другие файлы исходного кода. Начиная с главы 6, мы уточняем квалификатором `std::` все имена классов стандартной библиотеки, функций и объектов (например, `std::cout`) — это считается хорошей практикой.

Функция setName

Функция `setName` (строки 10–12) получает объект `string_view`, представляющий имя клиента, и присваивает аргумент `name` элементу данных `m_name`. Напомним, что `string_view` — это представление последовательности символов (современной строки `std::string` или традиционной строки в стиле C), доступное только для чтения. В строке 11 символы из параметра `name` копируются в `m_name`. Благодаря префиксу `m_` в имени `m_name`, параметр функции `name` легко отличить от элемента данных `m_name`.

Функция getName

Функция `getName` (строки 15–17) не имеет параметров и возвращает значение `m_name` из конкретного объекта класса `Account` в точку вызова в виде `const std::string&`. Поскольку возвращаемое значение является константной ссылкой, вы можете быть уверены, что вызывающий код не сможет изменять данные через эту ссылку.

Функции класса, объявленные с квалификатором const

Обратите внимание на `const` справа от списка параметров в заголовке `getName` (строка 15). При возврате `m_name` функция `getName` не изменяет и не должна изменять объект, для которого она вызывается. Объявление функции с квалификатором `const` сообщает компилятору: «Эта функция не должна изменять объект,

а если попробует, надо прервать компиляцию из-за ошибки». Это поможет вам сразу обнаружить ошибку, если вы случайно напишете код, который попытается некорректно изменить объект. Это объявление также сообщает компилятору, что функция `getName` может быть вызвана для объекта `const Account`, а также через ссылку или указатель на `const Account`.

9.3.2. Спецификаторы доступа private и public

Ключевое слово `private` (строка 18) — это **спецификатор доступа (access specifier)**. Все спецификаторы доступа обязательно дополняются двоеточием (например, `private:`). Элемент данных `m_name` (строка 19) объявлен после `private:`, поэтому является **скрытым (private)**, то есть доступным только для функций своего класса `Account`[1]. Когда вы объявляете элементы данных класса скрытыми, это называется **сокрытием данных (information hiding)** или **сокрытием деталей реализации (hiding implementation details)**. Сокрытие данных рекомендуется в C++ Core Guidelines[2] в качестве обычной практики и повсеместно применяется в объектно-ориентированном программировании. Поскольку элемент данных `m_name` инкапсулирован (скрыт), его могут использовать только функции класса `Account`: `setName` и `getName`. Как правило, все элементы данных надо объявлять после спецификатора доступа `private:`[3].

В классе `Account` также есть спецификатор доступа `public` (строка 8):

```
public:
```

Элементы класса, перечисленные после спецификатора доступа `public` и перед следующим спецификатором доступа, если он есть, являются **открытыми (public)**, то есть общедоступными. Их можно использовать везде, где объект класса находится в области видимости.

Объявление элементов данных скрытыми (`private`) упрощает отладку кода, потому что все проблемы с обработкой данных локализуются внутри функций класса. Подробнее о преимуществах `private` мы расскажем в разделе 9.7. В главе 10 мы рассмотрим спецификатор доступа `protected`.

Доступ к элементам класса по умолчанию

По умолчанию все элементы любого класса являются скрытыми (`private`), если вы не укажете иное. После строки со спецификатором доступа каждый объявленный элемент класса будет иметь этот уровень доступа — до тех пор, пока вы не укажете другой спецификатор доступа. Спецификаторы `public` и `private` можно повторять в разных местах класса, но это затрудняет понимание кода. Мы предпочитаем

---

[1] А также для дружественных функций класса, как вы увидите в разделе 9.18.

[2] C++ Core Guidelines, «C.9: Minimize Exposure of Members». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-private.

[3] Далее мы обычно будем опускать двоеточие, когда ключевые слова private и public упоминаются вне кода, как в этой сноске.

включать в каждый класс только один спецификатор `public`, группируя после него все открытые элементы, и только один спецификатор `private`, группируя после него все скрытые элементы.

9.4. Конструктор класса Account

Как было показано в разделе 9.3, при создании объекта класса `Account` его элемент `m_name` по умолчанию инициализируется пустой строкой. Но что делать, если при создании объекта класса `Account` вам нужно инициализировать элемент каким-то определенным значением, например указать имя клиента банка? В каждом классе можно определять **конструкторы (constructors)**, позволяющие инициализировать объекты этого класса нужными значениями. Конструктор — это специальная функция класса, которая должна иметь то же имя, что и класс. Конструкторы не могут возвращать значения, поэтому у них нет возвращаемого типа (даже `void`). C++ гарантирует вызов конструктора при создании объекта, поэтому конструктор идеально подходит для инициализации элементов данных объекта. Каждый раз, когда вы создавали какой-либо объект в наших примерах кода, для инициализации этого объекта вызывался конструктор соответствующего класса. Как вы скоро увидите, классы могут иметь перегруженные конструкторы.

Подобно обычным функциям класса, конструкторы могут иметь параметры. Соответствующие параметрам значения аргументов помогают вам инициализировать элементы данных объекта. Например, при создании объекта класса `Account` вы можете указать имя клиента банка, как это сделано в строке 10 на илл. 9.4:

```
Account account1{"Jane Green"};
```

Здесь строка `"Jane Green"` передается конструктору класса `Account` для инициализации данных объекта `account1`. Эта инструкция предполагает, что в классе `Account` есть конструктор, который может принимать строковый аргумент.

Определение класса Account

На илл. 9.3 показан класс `Account` с конструктором, получающим параметр `accountName` для инициализации элемента данных `m_name` при создании объекта `Account`.

```
1 // Fig. 9.3: Account.h
2 // Класс Account с конструктором, инициализирующим объект именем клиента
3 #include <string>
4 #include <string_view>
5
6 class Account {
7 public:
8    // Конструктор инициализирует элемент данных m_name параметром name
9    explicit Account(std::string_view name)
10       : m_name{name} { // Инициализируем элемент данных
11       // Пустое тело
12    }
```

Окончание ⇨

```
13
14    // Set-функция, записывающая значение элемента данных
15    void setName(std::string_view name) {
16       m_name = name;
17    }
18
19    // Get-функция, читающая значение элемента данных
20    const std::string& getName() const {
21       return m_name;
22    }
23 private:
24    std::string m_name; // Имя клиента банка
25 }; // Конец класса Account
```

Илл. 9.3 | Класс Account с конструктором, инициализирующим объект именем клиента банка

Определение конструктора в классе Account

В строках 9–12 на илл. 9.3 определяется конструктор класса `Account`. Обычно конструкторы являются открытыми (`public`), поэтому любой код, имеющий доступ к определению класса, может создавать и инициализировать объекты этого класса. В строке 9 указано, что параметр конструктора имеет тип `string_view`. При создании нового объекта класса `Account` вы передаете конструктору имя клиента в параметр типа `string_view`. Затем конструктор инициализирует элемент данных `m_name` значением этого параметра. В строке 9 на илл. 9.3 не указан тип возвращаемого значения (даже `void`), поскольку конструкторы не могут возвращать значения. Кроме того, конструкторы не могут быть объявлены как `const`, потому что инициализация объекта должна его изменять.

В конструкторе есть **список инициализаторов элементов (member-initializer list)**, который инициализирует элемент данных `m_name` (строка 10):

```
: m_name{name}
```

Инициализаторы элементов данных находятся между списком параметров конструктора и левой фигурной скобкой, начинающей тело конструктора. Список инициализаторов отделяется от списка параметров двоеточием (`:`).

Каждый инициализатор состоит из имени переменной элемента данных и последующих фигурных скобок, содержащих инициализирующее значение[1]. В данном случае инициализатор вызывает конструктор класса `std::string`, получающий значение `string_view`. Если класс содержит более одного элемента данных, их инициализаторы разделяются запятыми. Элементы данных класса инициализируются конструктором в том порядке, как они объявлены. Для наглядности кода перечисляйте инициализаторы в списке в том же порядке. Список инициализаторов элементов выполняется перед телом конструктора.

[1] В некоторых случаях фигурные скобки заменяются круглыми, например, при инициализации вектора заданного размера (илл. 6.14).

Хотя вы можете инициализировать элементы данных инструкциями присваивания в теле конструктора, в C++ Core Guidelines рекомендуется использовать инициализаторы[1]. Позже вы увидите, что инициализаторы могут быть эффективнее, чем инструкции присваивания. Кроме того, вы увидите, что некоторые элементы данных надо инициализировать только инициализаторами, поскольку им нельзя присваивать значения в теле конструктора.

Ключевое слово explicit

Объявляйте конструктор с ключевым словом `explicit`, если этот конструктор можно вызвать с одним аргументом (если у него только один параметр или если все параметры, кроме первого, имеют аргументы по умолчанию). Это не позволяет компилятору использовать конструктор для неявных преобразований типов[2]. Ключевое слово `explicit` означает, что конструктор класса `Account` должен быть вызван явно, как в строке 10 на илл. 9.4:

```
Account account1{"Jane Green"};
```

Пока просто запомните, что конструкторы с одним параметром надо объявлять как вызываемые явно, то есть с ключевым словом `explicit`. В разделе 11.9 вы увидите, что конструкторы с одним параметром, не объявленные как `explicit`, могут вызываться компилятором неявно для преобразования типов. Такие неявные вызовы конструкторов нежелательны, поскольку могут приводить к скрытым ошибкам.

Инициализация объектов класса Account при их создании

Программа `AccountTest` (илл. 9.4) инициализирует два объекта класса `Account` с помощью конструктора. В строке 10 создается объект класса `Account` с именем `account1`:

```
Account account1{"Jane Green"};
```

Эта инструкция вызывает конструктор класса `Account` (строки 9–12 на илл. 9.3), который использует аргумент `"Jane Green"` для инициализации элемента данных `m_name` в новом объекте. В строке 11 на илл. 9.4 процесс повторяется: аргумент `"John Blue"` передается для инициализации элемента данных `m_name` в объекте `account2`:

```
Account account2{"John Blue"};
```

В строках 14–16 мы вызываем функцию `getName` для каждого объекта класса `Account`, чтобы получить его имя и тем самым подтвердить корректность инициализации. Выходные данные показывают, что имена клиентов в объектах различаются. Каждый объект имеет свою собственную копию элемента `m_name`.

---

[1] C++ Core Guidelines, «C.49: Prefer Initialization to Assignment in Constructors». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-initialize.

[2] C++ Core Guidelines, «C.46: By Default, Declare Single-Argument Constructors explicit». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-explicit.

```
// Fig. 9.4: AccountTest.cpp
// Инициализация элемента m_name конструктором класса Account
// при создании объектов этого класса
#include <fmt/format.h>
#include <iostream>
#include "Account.h"

int main() {
   // Создаем два объекта класса Account
   Account account1{"Jane Green"};
   Account account2{"John Blue"};

   // Отображаем имена клиентов, хранящиеся в объектах
   std::cout << fmt::format(
      "account1 name is: {}\naccount2 name is: {}\n",
      account1.getName(), account2.getName());
}
```

```
account1 name is: Jane Green
account2 name is: John Blue
```

Илл. 9.4 | Инициализация элемента m_name конструктором класса Account при создании объектов этого класса

Конструктор по умолчанию

Напомним, что в строке 9 на илл. 9.1 объект класса `Account` создавался с пустыми фигурными скобками справа от имени переменной объекта:

```
Account myAccount{};
```

При выполнении этой инструкции C++ неявно вызывает для класса `Account` **конструктор по умолчанию (default constructor)**. Если в классе нет собственного конструктора, компилятор генерирует для этого класса конструктор по умолчанию, не имеющий параметров.

Конструктор по умолчанию не инициализирует элементы данных фундаментальных типов, но вызывает конструкторы по умолчанию для всех элементов данных, которые являются объектами других классов. Например, при создании объекта нашего первого класса `Account` (илл. 9.2) вызывается конструктор класса `std::string` по умолчанию, инициализирующий элемент данных `m_name` пустой строкой (`""`). Вы не видите этого в исходном коде, но именно так создаются объекты.

Когда вы объявляете объект с пустыми фигурными скобками, как в предыдущей инструкции, все элементы данных, не инициализированные явно, становятся **инициализированными значением по умолчанию (value initialized)**. Это означает, что они инициализируются нулями, а затем для каждого объекта вызывается конструктор по умолчанию. Если фигурных скобок нет, такие элементы данных заполняются неопределенными («мусорными») значениями.

Для класса с собственным конструктором не создается конструктор по умолчанию

Если вы определили в классе конструктор, компилятор не предоставит для этого класса конструктор по умолчанию. Например, вы не сможете создать объект класса `Account`, вызвав конструктор без аргументов (за исключением того случая, когда список параметров вашего конструктора пуст или у всех параметров есть аргументы по умолчанию). Позже мы покажем, как «заставить» компилятор создавать конструкторы по умолчанию даже для классов, имеющих собственные конструкторы. Если вас не устраивает инициализация элементов данных вашего класса значениями по умолчанию, инициализируйте каждый элемент в его объявлении или определите в классе такой конструктор, который инициализирует все элементы нужными значениями.

Специальные функции класса

Кроме конструктора по умолчанию, компилятор может генерировать несколько **специальных функций класса (special member functions)** в версиях по умолчанию: копирующий конструктор, перемещающий конструктор, оператор копирующего присваивания, оператор перемещающего присваивания и деструктор; а также новый оператор трехстороннего сравнения (C++20). В этой главе мы кратко представим вам копирующий конструктор, оператор копирующего присваивания и деструктор. В главе 11 мы расскажем об операторе трехстороннего сравнения и детальнее рассмотрим все специальные функции класса, в том числе:

- перечислим ситуации, в которых вам надо будет определять ваши собственные версии этих функций;
- укажем на рекомендации C++ Core Guidelines, относящиеся к этим функциям.

Вы увидите, что обычно должны проектировать классы таким образом, чтобы компилятор автоматически генерировал для вас эти специальные функции. Если вы не включаете в определение класса никакие специальные функции, это называется «правилом нуля».

9.5. Разработка программ с set-функциями и get-функциями

Как вы увидите в следующем разделе, *set*-функции класса (предназначенные для записи) и *get*-функции класса (предназначенные для чтения) могут контролировать все попытки записи в скрытые элементы данных объекта, а также способы передачи прочитанных данных в точку вызова. Это дает большие преимущества при разработке программ.

Клиентом (client) класса называется любой код, вызывающий функции этого класса. Если элемент данных открыт (объявлен как `public`), любой клиент может его увидеть и сделать с ним что угодно, например внести недопустимое изменение.

Если элемент данных скрыт (объявлен как `private`), клиент не может получить к нему прямой доступ. Но с первого взгляда кажется, что клиент все равно может делать с ним что угодно, вызывая доступные ему *set*-функции и *get*-функции класса. Кажется, что ничто не мешает клиенту в любой момент заглядывать извне в скрытые данные объекта (и видеть, как они хранятся в объекте) с помощью *get*-функций и произвольно изменять эти данные с помощью *set*-функций.

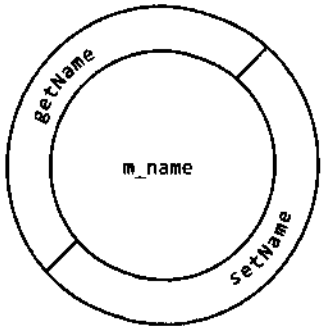
На самом деле *set*-функции могут быть написаны так, чтобы они проверяли получаемые аргументы и отклоняли все попытки присвоить элементам данных класса недопустимые значения. Примеры недопустимых аргументов:

- отрицательная температура тела человека;
- день в марте, выходящий за пределы диапазона с `1` по `31`;
- артикул товара, которого нет в каталоге продукции компании.

В свою очередь, *get*-функции могут возвращать данные в форматах, которые отличаются от исходных форматов элементов данных. В этом случае фактическое представление данных объекта остается скрытым от пользователя (от клиента класса). Например, класс `Grade` хранит оценку студента в виде числа `int` в диапазоне от `0` до `100`, но функция этого класса `getGrade` возвращает буквенную оценку в виде строки: `"A"` — для оценок от `90` до `100`, `"B"` для оценок от `80` до `89` и т. д. Внимательно контролируя обращения к скрытым (`private`) данным и способы их представления в возвращаемых значениях, вы можете свести к минимуму риск ошибок и сделать ваши программы надежными, безопасными и удобными для пользователей.

Концепция объекта класса Account с инкапсулированными данными

Представьте себе объект класса `Account` так, как показано на следующей диаграмме. Элемент данных `m_name`, изображенный в виде внутреннего круга, скрыт внутри объекта (объявлен как `private`). Получить к нему доступ можно только через внешний уровень открытых функций `getName` и `setName` (объявленных как `public`), изображенный в виде кольца. Любой клиент, которому надо взаимодействовать с объектом класса `Account`, может делать это только одним способом: вызывать функции внешнего уровня.



Как правило, все элементы данных класса объявляются скрытыми (`private`), а все предоставляемые клиентам функции класса объявляются открытыми (`public`). Позже мы рассмотрим исключения из этого правила: иногда некоторые элементы данных вашего класса могут быть открытыми, а некоторые функции — скрытыми. Открытые *set*-функции и *get*-функции, контролирующие доступ к скрытым данным, упрощают понимание кода и расширение функциональности программы в будущем. Постоянное внесение изменений в программы — это в ИТ-отрасли скорее правило, чем исключение. Старайтесь всегда исходить из того, что ваш код будет модифицироваться, причем довольно часто.

9.6. Добавление «денежной» переменной в класс Account

На илл. 9.5 показана следующая версия нашего класса `Account`. Теперь в нем есть два взаимосвязанных элемента данных: имя клиента банка и количество денег на его банковском счете (баланс). Согласно C++ Core Guidelines, все взаимосвязанные элементы данных рекомендуется определять в одном классе (или, как будет показано в разделе 9.21, — в одной структуре)[1].

```
// Fig. 9.5: Account.h
// Класс Account с элементами данных m_name и m_balance,
// с конструктором и функцией deposit, проверяющей данные
#include <algorithm>
#include <string>
#include <string_view>

class Account {
public:
   // Конструктор класса Account с двумя параметрами
   Account(std::string_view name, double balance)
      : m_name{name}, m_balance{std::max(0.0, balance)} { // Инициализация
      // Пустое тело
   }

   // Функция, пополняющая счет только после проверки данных
   void deposit(double amount) {
      if (amount > 0.0) { // Если значение допустимо
         m_balance += amount; // Прибавляем его к m_balance
      }
   }

   // Функция, возвращающая баланс счета
   double getBalance() const {
      return m_balance;
   }
```

Окончание ⇨

[1] C++ Core Guidelines, «C.1: Organize Related Data into Structures (structs or classes)». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-org.

```
27
28    // Функция, изменяющая имя клиента банка
29    void setName(std::string_view name) {
30       m_name = name; // Присваиваем значение name элементу данных m_name
31    }
32
33    // Функция, возвращающая имя клиента банка
34    const std::string& getName() const {
35       return m_name;
36    }
37 private:
38    std::string m_name;
39    double m_balance;
40 }; // Конец класса Account
```

Илл. 9.5 | Класс Account с элементами данных m_name и m_balance, с конструктором и функцией deposit, проверяющей данные

Элемент данных balance

Банк обслуживает множество счетов, каждый из который имеет собственный баланс. Каждый объект класса `Account` имеет собственные элементы данных `m_name` и `m_balance`. В строке 39 объявляется «денежный» элемент данных `m_balance` типа `double`[1].

Конструктор с двумя параметрами

В этом классе есть один конструктор и четыре функции. Обычно клиент банка, открывающий счет, сразу же вносит деньги, поэтому конструктор (в строках 11–14) получает второй параметр `balance` (`double`), который представляет первоначальный баланс. Мы не объявляли этот конструктор как `explicit`, поскольку его нельзя вызвать только с одним параметром.

Инициализатор элемента данных `m_balance` вызывает функцию `std::max` (заголовок `<algorithm>`), которая инициализирует `m_balance` числом `0.0` или параметром `balance`, если он больше. Это гарантирует, что при создании каждого объекта класса `Account` переменная `m_balance` будет иметь допустимое неотрицательное значение. Позже мы добавим проверку аргументов в теле конструктора и будем использовать исключения для обнаружения недопустимых аргументов.

Функция deposit

Функция `deposit` (строки 17–21) получает параметр `amount` типа `double` и не возвращает никакого значения. Инструкция в строках 18–20 гарантирует, что значение параметра `amount` будет прибавлено к `m_balance` только при условии, что оно больше нуля (то есть если сумма на счете корректна).

---

[1] Напомним, что в финансовых приложениях, предназначенных для практического применения, нельзя использовать тип double для представления денежных величин.

Функция getBalance

Функция `getBalance` (строки 24–26) позволяет клиентам класса `Account` получать значение `m_balance` из любого конкретного объекта этого класса. Функция имеет возвращаемый тип `double` и пустой список параметров. Как и `getName`, функция `getBalance` объявляется с квалификатором `const`, потому что возврат значения `m_balance` не изменяет и не должен изменять объекты класса `Account`.

Операции с объектами класса Account

Функция `main` на илл. 9.6 создает два объекта класса `Account` (строки 8–9). Они инициализируются допустимым (`50.00`) и недопустимым (`-7.00`) значениями баланса соответственно. В строках 12–15 мы выводим на экран имя и баланс, хранящиеся в объектах, вызывая их функции `getName` и `getBalance`. Наш класс гарантирует, что первоначальный баланс на каждом счете не может быть меньше нуля. Баланс объекта `account2` равен `0.0`, поскольку конструктор отклонил попытку ввода отрицательного числа, заменив его на `0.0`.

```
1 // Fig. 9.6: AccountTest.cpp
2 // Отображение и пополнение баланса объектов класса Account
3 #include <fmt/format.h>
4 #include <iostream>
5 #include "Account.h"
6
7 int main() {
8    Account account1{"Jane Green", 50.00};
9    Account account2{"John Blue", -7.00};
10
11    // Выводим на экран первоначальный баланс каждого счета
12    std::cout << fmt::format("account1: {} balance is ${:.2f}\n",
13       account1.getName(), account1.getBalance());
14    std::cout << fmt::format("account2: {} balance is ${:.2f}\n\n",
15       account2.getName(), account2.getBalance());
16
```

```
account1: Jane Green balance is $50.00
account2: John Blue balance is $0.00
```

Илл. 9.6 | Отображение и пополнение баланса объектов класса Account

Внесение «денег» на счет клиента банка

В строках 17–21 запрашивается ввод с клавиатуры числа, обозначающего пополнение баланса в объекте `account1`. В строке 22 вызывается функция `deposit` объекта `account1`, прибавляющая к его балансу значение аргумента `amount`. В строках 25–28 на экран выводятся данные из обоих объектов класса `Amount`: имя и баланс каждого клиента. Мы видим, что изменился только счет первого клиента (`amount1`).

```
17    std::cout << "Enter deposit amount for account1: "; // Ввод данных
18    double amount;
```

Окончание ⇨

```
19    std::cin >> amount; // Получаем число от пользователя
20    std::cout << fmt::format(
21       "adding ${:.2f} to account1 balance\n\n", amount);
22    account1.deposit(amount); // Прибавляем это число к балансу счета account1
23
24    // Выводим на экран баланс каждого счета
25    std::cout << fmt::format("account1: {} balance is ${:.2f}\n",
26       account1.getName(), account1.getBalance());
27    std::cout << fmt::format("account2: {} balance is ${:.2f}\n\n",
28       account2.getName(), account2.getBalance());
29
```

```
Enter deposit amount for account1: 25.37
adding $25.37 to account1 balance

account1: Jane Green balance is $75.37
account2: John Blue balance is $0.00
```

В строках 30–33 запрашивается число, которое будет прибавлено к балансу счета `account2`. В строке 34 вызывается функция `deposit` объекта `account2`, прибавляющая к его балансу значение аргумента `amount`. В строках 37–40 на экран снова выводятся данные из обоих объектов класса `Amount`. Мы видим, что теперь изменился только счет второго клиента (`amount2`).

```
30    std::cout << "Enter deposit amount for account2: "; // Ввод данных
31    std::cin >> amount; // Получаем число от пользователя
32    std::cout << fmt::format(
33       "adding ${:.2f} to account2 balance\n\n", amount);
34    account2.deposit(amount); // Прибавляем это число к балансу счета account2
35
36    // Выводим на экран баланс каждого счета
37    std::cout << fmt::format("account1: {} balance is ${:.2f}\n",
38       account1.getName(), account1.getBalance());
39    std::cout << fmt::format("account2: {} balance is ${:.2f}\n",
40       account2.getName(), account2.getBalance());
41 }
```

```
Enter deposit amount for account2: 123.45
adding $123.45 to account2 balance

account1: Jane Green balance is $75.37
account2: John Blue balance is $123.45
```

9.7. Отделение интерфейса от реализации на примере класса Time

В предыдущих примерах мы помещали определение класса в заголовок для повторного использования, а затем включали заголовок в файл исходного кода с функцией `main`, чтобы создавать объекты этого класса и управлять ими. К со-

жалению, размещение полного определения класса в заголовке раскрывает всю реализацию класса (его внутреннее устройство) клиентам класса. Заголовок — это обычный текстовый файл, который легко открыть и прочитать.

Общепринятая мудрость профессиональных программистов гласит, что для использования объектов любого класса его клиенты (например, функция `main`) должны знать совсем немного:

- какие функции этих объектов можно вызывать;
- какие аргументы надо предоставлять каждой функции;
- какой возвращаемый тип ожидается от каждой функции.

Клиентскому коду не обязательно знать о том, как именно реализованы функции класса. Это еще один пример принципа наименьших привилегий.

Если программист клиентского кода знает, как реализован класс, он может сделать свой код зависимым от каких-то особенностей этой реализации. И если в будущем реализация этого класса изменится, возникнут проблемы. Клиентский код надо писать так, чтобы его не приходилось переделывать в случае изменений в реализации класса. Сокрытие деталей реализации класса позволяет ее модернизировать, но при этом исключить (или хотя бы свести к минимуму) необходимость изменений в клиентском коде.

В следующем примере мы создадим и используем объект класса `Time`[1]. Мы продемонстрируем две важные концепции разработки программ на C++:

- Интерфейс класса должен быть отделен от реализации класса.
- В каждом заголовке необходима директива препроцессора `#pragma once`, запрещающая многократное (более одного раза) включение этого заголовка в какой-либо файл исходного кода. Поскольку каждый класс должен быть определен в программе только один раз, директива `#pragma once` предотвращает ошибки, вызванные повторяющимися определениями.

Модули (C++20) дают новые возможности отделения интерфейса от реализации

Как вы увидите в главе 16, модули, которые появились в C++20[2], позволяют обойтись без директив препроцессора, в том числе без директивы `#pragma once`. Вы также увидите, что с помощью модулей вы сможете отделять интерфейс от реализации разными способами: или в одном общем файле исходного кода, или в нескольких файлах.

---

[1] Это учебный пример. Для работы со значениями времени и даты вам не придется писать собственные классы; используйте заголовок <chrono> из стандартной библиотеки C++ (https://en.cppreference.com/w/cpp/chrono).

[2] На момент написания книги модули C++20 еще не были полностью реализованы в трех используемых нами компиляторах, поэтому мы рассмотрим модули в отдельной главе.

9.7.1. Интерфейс класса

Интерфейс (interface) определяет и стандартизирует способы взаимодействия человека с какой-либо системой. Например, элементы управления радиоприемником являются интерфейсом между его внутренними компонентами и пользователем. Через этот интерфейс пользователь может выполнять только ограниченный набор операций (таких, как настройка на радиостанцию, регулировка громкости, переключение между диапазонами AM и FM). Каждая операция может быть реализована по-разному: первый приемник реагирует на нажатие кнопки, второй — на поворот ручки, третий — на голосовую команду. Интерфейс определяет, какие операции приемник разрешает выполнять своим пользователям. Но интерфейс не определяет, как эти операции реализованы внутри приемника.

Аналогично интерфейс класса определяет, какие сервисы этот класс предоставляет своим клиентам и каким образом они должны запрашивать эти сервисы. Но интерфейс класса ничего не говорит о том, как эти сервисы реализованы внутри класса. Открытый (`public`) интерфейс класса состоит из открытых функций класса, которые также называют **открытыми сервисами (publuc services)** класса. Как вы скоро увидите, для создания интерфейса класса вам достаточно написать такое определение класса, в котором перечислены только прототипы его функций из секции `public`.

9.7.2. Отделение интерфейса класса от реализации класса

Чтобы отделить интерфейс класса от его реализации, мы разделили наш класс `Time` на два файла. В файле `Time.h` (илл. 9.7) определен класс `Time`, а в файле `Time.cpp` (илл. 9.8) определены функции класса `Time`. Такое разделение:

- позволяет использовать класс многократно;
- гарантирует, что клиенты класса знают о том, какие функции предоставляет класс, как их вызывать и какие возвращаемые типы данных ожидать;
- позволяет клиентам игнорировать реализацию (внутреннее устройство) функций класса.

Помимо вышеперечисленных преимуществ, такое разделение ускоряет компиляцию, поскольку файл с определениями функций (файл реализации) можно скомпилировать только один раз и не компилировать повторно, если реализация не изменяется.

По общепринятому соглашению определения функций класса помещаются в файл с расширением `.cpp`, имеющий то же имя, что и заголовок класса (например, `Time`). Некоторые компиляторы поддерживают и другие расширения имен файлов. На илл. 9.9 показана программа с функцией `main`, которая использует объекты нашего класса `Time`.

9.7.3. Определение класса

Заголовок `Time.h` (илл. 9.7) содержит определение класса `Time` (строки 8–17). Вместо определений функций здесь есть только их прототипы (строки 10–12), которые описывают открытый (`public`) интерфейс класса, не раскрывая реализации его функций. Прототип функции `setTime` в строке 10 указывает, что она принимает три аргумента `int` и возвращает значение `void`. Прототипы функций `to24HourString` и `to12HourString` (строки 11–12) указывают, что они не принимают аргументы и возвращают значение `string`. Если бы в классе были конструкторы, их тоже надо было бы объявить в заголовке, как мы сделаем в последующих примерах.

```
1 // Fig. 9.7: Time.h
2 // Определение класса Time
3 // Функции этого класса определены в файле Time.cpp
4 #pragma once // Предотвращаем повторные включения заголовка
5 #include <string>
6
7 // Определение класса Time
8 class Time {
9 public:
10    void setTime(int hour, int minute, int second);
11    std::string to24HourString() const; // 24-часовой формат строки
12    std::string to12HourString() const; // 12-часовой формат строки
13 private:
14    int m_hour{0}; // 0 - 23 (24-часовой формат времени)
15    int m_minute{0}; // 0 - 59
16    int m_second{0}; // 0 - 59
17 };
```

Илл. 9.7 | Определение класса Time

Заголовок по-прежнему содержит скрытые (`private`) элементы данных класса (строки 14–16). Каждый из них инициализируется значением `0` с помощью **внутриклассового инициализатора (in-class initializer)** (C++11). Компилятор должен знать обо всех элементах данных класса, чтобы зарезервировать соответствующий объем памяти для каждого из объектов класса. Включение заголовка `Time.h` в клиентский код предоставляет компилятору всю информацию, которая ему нужна, чтобы правильно вызывать из клиентского кода функции класса `Time`.

В C++ Core Guidelines рекомендуется использовать внутриклассовые инициализаторы, если элементы данных должны быть инициализированы константами[1]. В C++ Core Guidelines также рекомендуется, когда это возможно, инициализировать все элементы данных внутриклассовыми инициализаторами и позволить

[1] C++ Core Guidelines, «C.48: Prefer In-Class Initializers to Member Initializers in Constructors for Constant Initializers». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-in-class-initializer.

компилятору создать конструктор по умолчанию. Конструктор по умолчанию, созданный компилятором, может быть более эффективным, чем тот, который вы напишете[1].

Директива #pragma once

Большие программы обычно содержат много заголовков, и каждый из них, в свою очередь, может содержать другие заголовки с определениями и объявлениями. Поэтому после обработки всего исходного кода программы препроцессором очень вероятны случайные повторные включения в нее одного и того же заголовка. Если в программе окажется несколько одинаковых определений, это может привести к ошибкам компиляции. Директива `#pragma once` (строка 4) предотвращает включение заголовка `Time.h` в файл исходного кода более одного раза. Это иногда называют **защитой включения заголовков (include guard)**. В главе 16 мы обсудим другой способ предотвращения таких проблем: с помощью модулей (C++20).

9.7.4. Функции класса Time

В файле `Time.cpp` (илл. 9.8) определены функции класса `Time`, объявленные в строках 10–12 заголовка `Time.h` (илл. 9.7). Для функций `to24HourString` и `to12HourString` ключевое слово `const` должно присутствовать как в прототипах функций (илл. 9.7, строки 11–12), так и в определениях функций (илл. 9.8, строки 23 и 29).

```
 1 // Fig. 9.8: Time.cpp
 2 // Определения функций класса Time
 3 #include <fmt/format.h>
 4 #include <stdexcept> // Для класса исключений invalid_argument
 5 #include <string>
 6 #include "Time.h" // Определение класса Time
 7
 8 // Устанавливаем новое значение времени, используя 24-часовой формат
 9 void Time::setTime(int hour, int minute, int second) {
10    // Проверяем часы, минуты, секунды
11    if ((hour < 0 || hour >= 24) || (minute < 0 || minute >= 60) ||
12       (second < 0 || second >= 60)) {
13       throw std::invalid_argument{
14          "hour, minute or second was out of range"};
15    }
16
17    m_hour = hour;
18    m_minute = minute;
19    m_second = second;
20 }
21
```

[1] C++ Core Guidelines, «C.45: Don't Define a Default Constructor That Only Initializes Data Members; Use In-Class Member Initializers Instead». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-default.

```
22 // Возвращаем время в 24-часовом формате (HH:MM:SS)
23 std::string Time::to24HourString() const {
24    return fmt::format("{:02d}:{:02d}:{:02d}", m_hour, m_minute, m_second);
25 }
26
27 // Возвращаем время в 12-часовом формате (HH:MM:SS AM или PM)
28 std::string Time::to12HourString() const {
29    return fmt::format("{}:{:02d}:{:02d} {}",
30       ((m_hour % 12 == 0) ? 12 : m_hour % 12), m_minute, m_second,
31       (m_hour < 12 ? "AM" : "PM"));
32 }
```

Илл. 9.8 | Определения функций класса Time

9.7.5. Включение заголовка класса в файл исходного кода

Чтобы указать, что функции в файле `Time.cpp` являются частью класса `Time`, мы должны включить в этот файл заголовок `Time.h` (илл. 9.8, строка 6). Это позволяет нам использовать имя класса `Time` в файле `Time.cpp` (строки 9, 23 и 28). При компиляции файла `Time.cpp` компилятор использует информацию из `Time.h`, чтобы гарантировать, что:

- первая строка каждой функции класса соответствует ее прототипу в файле `Time.h`;
- каждая функция класса знает об элементах данных класса и о других функциях класса.

9.7.6. Оператор разрешения области видимости (::)

Имя каждой функции (строки 9, 23 и 28) начинается с имени класса и оператора `::` (двойного двоеточия), который называется **оператором разрешения области видимости (scope resolution operator)**. Это «привязывает» функции к определению класса `Time`, которое теперь находится в отдельном файле (илл. 9.7). Префикс `Time::` перед именем функции сообщает компилятору, что функция находится в области видимости класса `Time` и ее имя известно другим элементам класса.

Без префикса `Time::` перед именем каждой функции компилятор рассматривал бы ее как глобальную функцию, не имеющую отношения к классу `Time`. Такие функции, иногда называемые **свободными функциями (free functions)**, не могут получать доступ к скрытым данным классов или вызывать функции классов без указания объекта. Таким образом, без префикса `Time::` компилятор не смог бы скомпилировать эти функции из-за отсутствия информации о том, что в классе `Time` объявлены переменные `m_hour`, `m_minute` и `m_second`. Возникли бы ошибки компиляции в строках 17–19, 24 и 30–31 (илл. 9.8), поскольку `m_hour`, `m_minute` и `m_second` не объявлены ни как локальные переменные в функциях, ни как глобальные переменные.

9.7.7. Функция setTime и исключения

Функция `setTime` (строки 9–20) — это открытая функция, которая объявляет три параметра `int` и использует их для установки времени. В строках 11–12 все три аргумента проверяются на соответствие допустимым диапазонам. Если условие выполняется, в строках 17–19 значения аргументов присваиваются элементам данных `m_hour`, `m_minute` и `m_second` соответственно. Аргумент `hour` должен быть больше или равен `0` и меньше `24`, поскольку 24-часовой формат представляет часы в виде целых чисел от `0` до `23`. Аналогично аргументы `minute` и `second` должны быть больше или равны `0` и меньше `60`.

Если какой-либо из аргументов выходит за пределы допустимого диапазона, в строках 13–14 функция `setTime` выдает исключение типа `invalid_argument` (заголовок `<stdexcept>`), сообщая клиентскому коду о недопустимом значении аргумента. Как было показано в разделе 6.15, для перехвата исключений и продолжения работы программы можно использовать инструкции `try...catch`, что мы и сделаем в следующем примере (илл. 9.9). Инструкция `throw` создает новый объект исключения типа `invalid_argument` и инициализирует его нашей строкой с сообщением об ошибке. После создания объекта исключения инструкция `throw` завершает работу функции `setTime`. Затем исключение возвращается в код, вызвавший функцию `setTime`.

Итак, объект класса `Time` не может содержать недопустимые значения времени, потому что:

- при создании объекта `Time` вызывается его конструктор по умолчанию и все элементы данных инициализируются числом `0`, как указано в строках 14–16 (илл. 9.7), что означает полночь (12 AM);
- все последующие попытки клиентского кода изменить элементы данных класса контролируются функцией `setTime`.

9.7.8. Функции to24HourString и to12HourString

Функция `to24HourString` (строки 23–25 на илл. 9.8) не принимает аргументы и возвращает строку, в которой время записано в 24-часовом формате, например `"13:30:07"`. Каждый заполнитель `{:02d}` форматирует целое число (`d`) в поле шириной `2` символа (строка 24). Цифра `0` в этом заполнителе сообщает компилятору, что перед каждым значением, содержащим менее двух цифр, надо добавить ноль.

Функция `to12HourString` (строки 28–32) не принимает аргументы и возвращает строку, в которой время записано в 12-часовом формате, например `"10:54:27 AM"` (AM — до полудня) или `"1:27:06 PM"` (PM — после полудня). Заполнитель `{:02d}` форматирует значения минут и секунд (`m_minute` и `m_second`) в поле шириной `2` символа, добавляя при необходимости начальный ноль. В строке 30 условный оператор `?:` используется для форматирования значения часов (`m_hour`). Если `m_hour` равно `0` (полночь) или `12` (полдень), отображается число `12`; в противном случае мы используем оператор остатка от деления `%`, чтобы получить число от

1 до 11. Условный оператор в строке 31 определяет, что надо добавить в конце: AM или PM.

9.7.9. Неявное встраивание функций

Если функция полностью определена в теле класса (как в наших примерах с классом `Account`), компилятор, скорее всего, неявно объявит ее как встроенную, или `inline`-функцию (раздел 5.11). В большинстве случаев это ускоряет работу программы. Но помните, что компилятор может и не делать этого. Оптимизирующий компилятор оставляет за собой право (но не обязан) встраивать даже те функции, которые вы объявили без ключевого слова `inline`, если у него есть доступ к их определениям.

В заголовке класса можно определять только самые простые и стабильные функции, реализация которых вряд ли изменится, а также те, которые должны работать максимально быстро. Дело в том, что каждое изменение в заголовке требует перекомпиляции всех файлов исходного кода, зависящих от этого заголовка. Это занимает очень много времени, если вы пишете большую программу.

9.7.10. Функции класса в сравнении с глобальными функциями

Функции класса `to24HourString` и `to12HourString` не принимают аргументы. Они неявно знают об элементах данных объекта `Time`, из которого вызываются, и могут получать к ним доступ. Это ценное преимущество объектно-ориентированного программирования: обычно функции классов не принимают аргументы или принимают меньше аргументов, чем функции в программах, которые не являются объектно-ориентированными. Это снижает вероятность передачи неправильных аргументов, неправильного количества аргументов или аргументов в неправильном порядке.

9.7.11. Использование класса Time

Когда класс `Time` определен, его можно использовать в качестве типа данных в объявлениях:

```
Time sunset{}; // Объект класса Time (тип объекта - Time)
std::array<Time, 5> arrayOfTimes{}; // Массив из 5 объектов Time
Time& dinnerTimeRef{sunset}; // Ссылка на объект Time
Time* timePtr{&sunset}; // Указатель на объект Time
```

На илл. 9.9 показан пример создания и использования объекта класса `Time`. Отделение интерфейса класса `Time` от реализации его функций не влияет на использование класса клиентским кодом. В строке 8 включен заголовок `Time.h`, поэтому компилятор знает, сколько памяти надо зарезервировать для объекта `t` (строка 17), и может гарантировать, что объекты класса `Time` создаются и обрабатываются в клиентском коде корректно.

```
// fig09_09.cpp
// Программа для тестирования класса Time
// Внимание: этот файл должен быть скомпонован с Time.cpp
#include <fmt/format.h>
#include <iostream>
#include <stdexcept> // Для класса исключений invalid_argument
#include <string_view>
#include "Time.h" // Определение класса Time в файле Time.h

// Отображаем время в 24-часовом и 12-часовом форматах
void displayTime(std::string_view message, const Time& time) {
   std::cout << fmt::format("{}\n24-hour time: {}\n12-hour time: {}\n\n",
      message, time.to24HourString(), time.to12HourString());
}

int main() {
   Time t{}; // Создаем объект t класса Time

   displayTime("Initial time:", t); // Отображаем начальное значение t
   t.setTime(13, 27, 6); // Изменяем время
   displayTime("After setTime:", t); // Отображаем новое значение t

   // Пробуем установить недопустимое время
   try {
      t.setTime(99, 99, 99); // Все три значения ошибочны!
   }
   catch (const std::invalid_argument& e) {
      std::cout << fmt::format("Exception: {}\n\n", e.what());
   }

   // Отображаем значение t после попытки установить недопустимое время
   displayTime("After attempting to set an invalid time:", t);
}
```

```
Initial time:
24-hour time: 00:00:00
12-hour time: 12:00:00 AM

After setTime:
24-hour time: 13:27:06
12-hour time: 1:27:06 PM

Exception: hour, minute and/or second was out of range

After attempting to set an invalid time:
24-hour time: 13:27:06
12-hour time: 1:27:06 PM
```

Илл. 9.9 | Программа для тестирования класса Time

На протяжении всей программы мы отображаем данные объекта `Time`, используя функцию `displayTime` (строки 11–14), которая вызывает функции класса `Time`:

`to24HourString` и `to12HourString`. В строке 17 создается объект `t`. Напомним, что в классе `Time` не определен пользовательский конструктор, поэтому вызывается конструктор по умолчанию, созданный компилятором. Таким образом, переменным `m_hour`, `m_minute` и `m_second` присваивается значение `0` через их инициализаторы в определении класса `Time`. В строке 19 мы отображаем время в 24-часовом и 12-часовом форматах, чтобы подтвердить правильность инициализации элементов. В строке 20 мы устанавливаем новое время, вызывая функцию `setTime`, а в строке 21 снова отображаем время в обоих форматах.

Вызов функции setTime с недопустимыми значениями

Чтобы подтвердить проверку аргументов функцией `setTime`, мы вызываем ее в строке 25 с недопустимыми аргументами `99` для всех трех параметров: `hour`, `minute` и `second`. Мы поместили эту инструкцию в блок `try` (строки 24–26), чтобы перехватывать исключение `invalid_argument`, выдаваемое функцией `setTime`. Когда возникает исключение, оно перехватывается в строках 27–29, а в строке 28 отображается сообщение об ошибке, полученное с помощью функции `what` класса `invalid_argument`. В строке 32 мы снова отображаем время, подтверждая, что вызов функции с ошибочными аргументами не повредил сохраненные в объекте данные.

9.7.12. Размер объекта

Новички в объектно-ориентированном программировании часто думают, что объекты должны быть довольно большими, поскольку они содержат элементы данных и функции класса. Это может показаться логичным. Вы можете представлять себе, что элементы данных и функции хранятся внутри каждого объекта (и наше обсуждение, безусловно, способствует подобному взгляду). Но на самом деле это не так.

Созданный в памяти компьютера объект содержит только элементы данных, но не функции класса. Код функций класса хранится в памяти отдельно от всех объектов класса. Каждому объекту нужны собственные элементы данных, потому что они должны быть независимыми от других объектов. Но код функций идентичен для всех объектов класса, поэтому нет необходимости повторять его в каждом объекте.

9.8. Процесс компиляции и компоновки программы

Часто интерфейс и реализация класса создаются одним программистом, а использует их другой программист, пишущий клиентский код. Например, **программист-разработчик класса (class-implementation programmer)**, ответственный за создание повторно используемого класса `Time`, пишет заголовок `Time.h` и файл исходного кода `Time.cpp`, содержащий этот заголовок. Затем он передает оба файла программисту клиентского кода. Исходный код повторно используемого класса

часто доступен программистам клиентского кода в виде библиотеки, которую они могут загрузить с веб-сайта, например с портала GitHub.

Для использования класса `Time` программист клиентского кода должен знать только интерфейс этого класса, а также иметь возможность скомпилировать файл `Time.cpp` и затем **скомпоновать (link)** готовое приложение. Поскольку интерфейс класса является частью определения класса в заголовке `Time.h`, программист клиентского кода должен включить этот заголовок в свой файл. Определение класса в заголовке `Time.h` позволяет компилятору убедиться, что клиентский код правильно создает и использует объекты класса `Time`.

В результате компиляции файлов исходного кода создаются файлы **объектного кода (object code)**, из которых **компоновщик (linker)** создает исполняемый файл, то есть готовое приложение. Чтобы получить самое простое приложение с классом `Time`, надо скомпоновать в единое целое:

- объектный код функции `main` (в которой находится клиентский код);
- объектный код функций класса `Time`;
- объектный код тех классов стандартной библиотеки C++ (например, `std::string`), которые использовались программистом класса и программистом клиентского кода.

Результатом компоновки всех компонентов программы из раздела 9.7 является готовое приложение, которое можно запускать для демонстрации работы класса `Time`. Как правило, компилятор или интегрированная среда разработки (IDE) после компиляции исходного кода автоматически вызывают компоновщик, создающий готовое приложение.

Компиляция программ, содержащих более одного файла исходного кода

В разделе 1.2 мы показали, как скомпилировать и запустить приложение, содержащее только один файл исходного кода (`.cpp`). Чтобы скомпилировать и скомпоновать приложение, содержащее несколько файлов исходного кода, выполните следующие действия[1]:

- Если вы работаете в Microsoft Visual Studio, добавьте в свой проект (как показано в разделе 1.2.1) все свои заголовки и файлы исходного кода, из которых состоит программа, затем создайте и запустите проект. Вы можете поместить заголовки в папку `Header Files` проекта, а файлы исходного кода — в папку `Source Files` проекта, но такое разделение файлов характерно только для больших проектов. Программы будут компилироваться и в том случае, если вы поместите все файлы в папку `Source Files`.
- Если вы работаете в командной строке с компилятором g++ или clang++, перейдите в каталог, содержащий все файлы вашей программы. Затем в команде компиляции перечислите имена всех файлов `.cpp` или используйте параметр `*.cpp` для компиляции всех имеющихся в этом каталоге файлов

[1] В главе 16 мы рассмотрим другой вариант, основанный на модулях C++20.

с расширением .cpp. Препроцессор автоматически найдет в этом каталоге все заголовки, относящиеся к вашей программе.

- Если вы работаете в Apple Xcode, добавьте в свой проект (как показано в разделе 1.2.2) все заголовки и файлы исходного кода, из которых состоит программа, затем создайте и запустите проект.

9.9. Область видимости класса и доступ к элементам класса

Элементы данных и функции класса находятся в области видимости этого класса. Функции, не являющиеся элементами классов (например, `main`), по умолчанию определяются в глобальной области видимости. В пределах области видимости класса для обращения к любому из его элементов достаточно указать его имя, поскольку каждый элемент непосредственно доступен всем функциям класса. Но для обращения к любому элементу за пределами области видимости класса, то есть к открытому (`public`) элементу другого класса, надо перед именем элемента указать **дескриптор (handle)** объекта, содержащего этот элемент. Дескриптором может быть:

- имя объекта;
- ссылка на объект;
- указатель на объект;
- указатель на элемент класса.

Тип дескриптора помогает компилятору идентифицировать интерфейс (то есть набор функций), доступный клиентам класса через этот дескриптор. В разделе 9.19 вы увидите, что компилятор автоматически добавляет неявный дескриптор (указатель `this`) каждый раз, когда вы обращаетесь из объекта к его элементу данных или к функции его класса.

Операторы «точка» (.) и «стрелка» (->)

Как вы уже знаете, для обращения к элементу объекта можно добавить после имени объекта или ссылки на объект оператор «точка» (`.`), а затем имя элемента. Похожим образом вы можете обращаться к элементу и через указатель на объект. Для этого добавьте после имени указателя оператор «стрелка» (`->`), а затем имя элемента: *ИмяУказателя->ИмяЭлемента*.

Доступ к открытым элементам класса через объекты, ссылки и указатели

Рассмотрим класс `Account`, в котором есть открытая функция `deposit`. Предположим, в нем есть такие объявления:

```
Account account{}; // Объект класса Account
Account& ref{account}; // Ссылка на объект класса Account
Account* ptr{&account}; // Указатель на объект класса Account
```

Вы можете вызвать функцию `deposit`, используя операторы «точка» (`.`) и «стрелка» (`->`):

```
account.deposit(123.45); // Вызов функции deposit через имя объекта
ref.deposit(123.45); // Вызов функции deposit через ссылку на объект
ptr->deposit(123.45); // Вызов функции deposit через указатель на объект
```

Напомним, что вы должны использовать ссылки вместо указателей, когда это возможно. Мы продолжим рассматривать указатели в тех случаях, когда они необходимы, а также для того, чтобы вы умели работать с ними, ведь они могут встретиться вам в устаревшем коде.

9.10. Функции доступа и служебные функции

Функции доступа

Функции доступа (access functions) могут получать или отображать данные, но не могут изменять их. Кроме того, они могут проверять различные условия, возвращая `true` или `false`. Такие функции часто называют **предикатными функциями (predicate functions)**. Примером может служить функция `empty` класса `std::array` или `std::vector`. Программа вызывает функцию `empty` перед попыткой прочитать элементы из контейнера[1].

Служебные функции

Служебные функции (utility functions), также называемые **вспомогательными функциями (helper functions)**, — это скрытые (`private`) функции, которые поддерживают работу других функций класса и не предназначены для использования клиентами класса. Как правило, служебная функция содержит код, который при ее отсутствии пришлось бы дублировать в основных функциях класса. Во многих кодовых базах существуют соглашения об именовании скрытых функций класса; например, их имена могут начинаться с символа подчеркивания (_).

9.11. Конструкторы с аргументами по умолчанию

На илл. 9.10–9.12 показан усовершенствованный код класса `Time`, содержащий конструктор с аргументами по умолчанию.

9.11.1. Усовершенствованный класс Time

Как и другие функции, конструкторы могут иметь аргументы по умолчанию. В строке 11 на илл. 9.10 объявляется конструктор класса `Time`, задающий для

[1] Многие программисты начинают имена предикатных функций со слова «is». Например, полезными предикатными функциями нашего класса Time могут быть isAM и isPM. Таким функциям должен предшествовать атрибут [[nodiscard]], требующий от компилятора убедиться, что возвращаемое значение используется в вызывающем коде, а не игнорируется.

каждого параметра аргумент по умолчанию `0`. Конструктор, в котором для всех параметров есть аргументы по умолчанию, является конструктором по умолчанию — он может быть вызван без аргументов[1]. В каждом классе может быть только один конструктор по умолчанию. Если изменилось значение по умолчанию какого-либо аргумента, после этого необходимо заново скомпилировать весь клиентский код, чтобы программа по-прежнему работала правильно.

```
1 // Fig. 9.10: Time.h
2 // Класс Time, содержащий конструктор с аргументами по умолчанию
3 // Функции класса определены в Time.cpp
4 #pragma once // Предотвращаем повторные включения заголовка
5 #include <string>
6
7 // Определение класса Time
8 class Time {
9 public:
10    // Это конструктор по умолчанию, так как он может быть вызван без аргументов
11    explicit Time(int hour = 0, int minute = 0, int second = 0);
12
13    // Set-функции (для изменения данных)
14    void setTime(int hour, int minute, int second);
15    void setHour(int hour); // Устанавливаем hour (после проверки)
16    void setMinute(int minute); // Устанавливаем minute (после проверки)
17    void setSecond(int second); // Устанавливаем second (после проверки)
18
19    // Get-функции (для получения данных)
20    int getHour() const; // Возвращаем hour
21    int getMinute() const; // Возвращаем minute
22    int getSecond() const; // Возвращаем second
23
24    std::string to24HourString() const; // 24-часовой формат строки
25    std::string to12HourString() const; // 12-часовой формат строки
26 private:
27    int m_hour{0}; // 0 - 23 (24-часовой формат часов)
28    int m_minute{0}; // 0 - 59
29    int m_second{0}; // 0 - 59
30 };
```

Илл. 9.10 | Класс Time, содержащий конструктор с аргументами по умолчанию

Конструктор класса Time

На илл. 9.11 в строках 9–11 определяется конструктор класса `Time`, который вызывает функцию `setTime` для проверки и изменения значений элементов данных. Функция `setTime` (строки 14–31) гарантирует, что `hour` находится в диапазоне `0-23`, а `minute` и `second` — в диапазоне `0-59`. Если какой-либо аргумент находится

[1] Далее вы увидите, что концепты (C++20, глава 15) позволяют вам перегружать любую функцию, в том числе функцию класса, для использования с типами, которые соответствуют вашим требованиям.

вне допустимого диапазона, функция `setTime` выдаст исключение и в программе не будет объекта `Time`, потому что конструктор не завершит его создание. В эту версию функции `setTime` мы включили отдельные инструкции `if`, чтобы проверять каждый аргумент и выдавать точные сообщения об ошибке, объясняющие, какой из аргументов находится вне допустимого диапазона. Каждая из функций `setHour`, `setMinute` и `setSecond` (строки 34–40) вызывает функцию `setTime`, передавая ей свой аргумент и текущие значения двух других элементов данных. Например, функция `setHour` передает свой аргумент `hour` для изменения `m_hour`, а также текущие значения `m_minute` и `m_second`.

```
 1 // Fig. 9.11: Time.cpp
 2 // Определения функций класса Time
 3 #include <fmt/format.h>
 4 #include <stdexcept>
 5 #include <string>
 6 #include "Time.h" // Определение класса Time
 7
 8 // Конструктор класса Time инициализирует каждый элемент данных
 9 Time::Time(int hour, int minute, int second) {
10    setTime(hour, minute, second);
11 }
12
13 // Устанавливаем новые значения, используя 24-часовой формат
14 void Time::setTime(int hour, int minute, int second) {
15    // Проверяем hour, minute и second
16    if (hour < 0 || hour >= 24) {
17       throw std::invalid_argument{"hour was out of range"};
18    }
19
20    if (minute < 0 || minute >= 60) {
21       throw std::invalid_argument{"minute was out of range"};
22    }
23
24    if (second < 0 || second >= 60) {
25       throw std::invalid_argument{"second was out of range"};
26    }
27
28    m_hour = hour;
29    m_minute = minute;
30    m_second = second;
31 }
32
33 // Устанавливаем значение hour
34 void Time::setHour(int hour) {setTime(hour, m_minute, m_second);}
35
36 // Устанавливаем значение minute
37 void Time::setMinute(int minute) {setTime(m_hour, minute, m_second);}
38
39 // Устанавливаем значение second
40 void Time::setSecond(int second) {setTime(m_hour, m_minute, second);}
41
```

```
42 // Возвращаем значение hour
43 int Time::getHour() const {return m_hour;}
44
45 // Возвращаем значение minute
46 int Time::getMinute() const {return m_minute;}
47
48 // Возвращаем значение second
49 int Time::getSecond() const {return m_second;}
50
51 // Возвращаем время в 24-часовом формате (HH:MM:SS)
52 std::string Time::to24HourString() const {
53    return fmt::format("{:02d}:{:02d}:{:02d}",
54       getHour(), getMinute(), getSecond());
55 }
56
57 // Возвращаем время в 12-часовом формате (HH:MM:SS AM или PM)
58 std::string Time::to12HourString() const {
59    return fmt::format("{}:{:02d}:{:02d} {}",
60       ((getHour() % 12 == 0) ? 12 : getHour() % 12),
61       getMinute(), getSecond(), (getHour() < 12 ? "AM" : "PM"));
62 }
```

Илл. 9.11 | Определения функций класса Time

В C++ Core Guidelines есть много рекомендаций по созданию конструкторов. Мы рассмотрим некоторые из них в следующих двух главах. Если класс содержит **инвариант класса (class invariant)**[1], то есть его элементы данных должны иметь заранее заданные значения или диапазоны значений (как в нашем классе `Time`), то в этом классе должен быть определен конструктор, который проверяет аргументы и в случае недопустимого значения выдает исключение, предотвращающее создание объекта[2,3,4].

Тестирование усовершенствованного класса Time

Функция `main` на илл. 9.12 инициализирует пять объектов класса `Time`:

- со всеми тремя аргументами по умолчанию, заданными неявным вызовом конструктора (строка 16);

---

[1] «Инвариант класса». Был доступен: 15 сентября 2023 г. https://ru.wikipedia.org/wiki/Инвариант_класса.

[2] C++ Core Guidelines, «C.40: Define a Constructor if a Class Has an Invariant». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-ctor.

[3] C++ Core Guidelines, «C.41: A Constructor Should Create a Fully Initialized Object». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-complete.

[4] C++ Core Guidelines, «C.42: If a Constructor Cannot Construct a Valid Object, Throw an Exception». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-throw.

- с одним аргументом (строка 17);
- с двумя аргументами (строка 18);
- с тремя аргументами (строка 19);
- с тремя недопустимыми аргументами (строка 29).

Программа отображает данные каждого объекта в 24-часовом и 12-часовом форматах. Для объекта `t5` (строка 29) программа выводит сообщение об ошибке, поскольку аргументы конструктора находятся вне допустимого диапазона. Переменная `t5` никогда не будет представлять полностью созданный объект в этой программе, потому что исключение выдается во время выполнения конструктора.

```
 1 // fig09_12.cpp
 2 // Конструктор с аргументами по умолчанию
 3 #include <fmt/format.h>
 4 #include <iostream>
 5 #include <stdexcept>
 6 #include <string>
 7 #include "Time.h" // Определение класса Time в файле Time.h
 8
 9 // Отображаем время в 24-часовом и 12-часовом форматах
10 void displayTime(std::string_view message, const Time& time) {
11    std::cout << fmt::format("{}\n24-hour time: {}\n12-hour time: {}\n\n",
12       message, time.to24HourString(), time.to12HourString());
13 }
14
15 int main() {
16    const Time t1{}; // Все аргументы по умолчанию
17    const Time t2{2}; // Указываем hour; minute и second - по умолчанию
18    const Time t3{21, 34}; // Указываем hour и minute; second - по умолчанию
19    const Time t4{12, 25, 42}; // Указываем все три аргумента
20
21    std::cout << "Constructed with:\n\n";
22    displayTime("t1: all arguments defaulted", t1);
23    displayTime("t2: hour specified; minute and second defaulted", t2);
24    displayTime("t3: hour and minute specified; second defaulted", t3);
25    displayTime("t4: hour, minute and second specified", t4);
26
27    // Пробуем создать объект t5 с недопустимыми аргументами
28    try {
29       const Time t5{27, 74, 99}; // Все аргументы ошибочны!
30    }
31    catch (const std::invalid_argument& e) {
32       std::cerr << fmt::format("t5 not created: {}\n", e.what());
33    }
34 }
```

```
Constructed with:

t1: all arguments defaulted
24-hour time: 00:00:00
```

```
12-hour time: 12:00:00 AM

t2: hour specified; minute and second defaulted
24-hour time: 02:00:00
12-hour time: 2:00:00 AM

t3: hour and minute specified; second defaulted
24-hour time: 21:34:00
12-hour time: 9:34:00 PM

t4: hour, minute and second specified
24-hour time: 12:25:42
12-hour time: 12:25:42 PM

t5 not created: hour was out of range
```

Илл. 9.12 | Конструктор с аргументами по умолчанию

Рекомендации разработчикам на примере get-функций, set-функций и конструктора класса Time

В теле класса `Time` неоднократно вызываются его *set*-функции и *get*-функции. В частности, конструктор (илл. 9.11, строки 9–11) вызывает `setTime`, а функции `to24HourString` и `to12HourString` вызывают функции `getHour`, `getMinute` и `getSecond` (строки 54 и 60–61). Во всех этих случаях мы получаем прямой доступ к скрытым (`private`) данным класса.

Во внутреннем представлении класса мы храним время в трех числах `int`, занимающих 12 байт памяти в системах с 4-байтовыми числами этого типа. Предположим, нам нужно представить время иначе: в виде количества прошедших с полуночи секунд, чтобы обойтись всего одним числом `int`, занимающим 4 байта. Для этого достаточно изменить только тела функций, которые напрямую обращаются к скрытым элементам данных. В этом классе изменения затронут только тела функции `setTime`, а также *set*-функций и *get*-функций для `m_hour`, `m_minute` и `m_second`. При этом не придется изменять ни конструктор класса, ни функции `24hourstring` и `12hourstring`, поскольку они не обращаются к данным напрямую.

Дублирование инструкций в нескольких функциях или конструкторах затрудняет модификацию представления данных внутри класса. Та реализация конструктора класса `Time` и функций `to24HourString` и `to12HourString`, которая показана в этом примере, снижает вероятность ошибок при изменении реализации класса.

У опытных программистов есть важный принцип: надо избегать повторения фрагментов кода. Этот принцип называется «Don't repeat yourself» («Не повторяйтесь»)[1]. Если какой-то фрагмент кода должен выполняться в нескольких местах программы, надо поместить его в функцию, которая может быть вызвана конструктором

[1] «Don't Repeat Yourself». Был доступен: 15 сентября 2023 г. https://ru.wikipedia.org/wiki/Don't_repeat_yourself.

класса и другими функциями. Это упрощает тестирование и модификацию кода, а также снижает вероятность ошибок при внесении любых изменений.

Конструктор может вызывать другие функции класса, но при этом нужна осторожность. Конструктор инициализирует объект, поэтому элементы данных, используемые в вызываемой функции, могут оказаться еще не инициализированными. Обращения к элементам данных до их инициализации приводят к логическим ошибкам.

Объявление элементов данных скрытыми (`private`) и последующий контроль доступа к ним (особенно перед изменением данных) с помощью открытых функций помогают обеспечить целостность данных. Все преимущества целостности данных не появятся сами собой, если вы просто объявите элементы данных скрытыми. Вы должны обеспечить еще и проверку корректности всех значений.

9.11.2. Перегрузка конструкторов и делегирование конструкторов (C++11)

В разделе 5.15 мы рассматривали перегруженные функции. Конструкторы и другие функции класса тоже можно перегружать. Перегруженные конструкторы позволяют инициализировать объекты с разными типами аргументов и (или) с разным количеством аргументов. Чтобы перегрузить конструктор, напишите прототип и определение для каждой перегруженной версии. Это также относится к перегруженным функциям класса.

На илл. 9.10–9.12 конструктор класса `Time` имел аргументы по умолчанию для всех параметров. Вместо этого конструктора мы могли бы определить четыре перегруженных конструктора со следующими прототипами:

```
Time(); // Аргументы по умолчанию для m_hour, m_minute и m_second: 0
explicit Time(int hour); // Аргументы по умолчанию для m_minute и m_second: 0
Time(int hour, int minute); // Аргументы по умолчанию для m_second: 0
Time(int hour, int minute, int second); // Нет аргументов по умолчанию
```

Конструктор может вызывать для выполнения своих задач не только другие функции класса, но и другие конструкторы своего класса. Вызывающий конструктор называется **делегирующим конструктором (delegating constructor)** (C++11), так как он делегирует свою работу другому конструктору. В C++ Core Guidelines рекомендуется определять общий код перегруженных конструкторов в одном конструкторе, а затем вызывать его из делегирующих конструкторов[1]. До C++11 общий код помещали в служебную функцию, вызываемую всеми конструкторами.

Выше мы показали прототипы четырех перегруженных конструкторов класса `Time`. Первые три из них могут делегировать работу одному конструктору с тремя аргументами `int`, передавая `0` в качестве значения по умолчанию для отсутствующих

[1] C++ Core Guidelines, «C.51: Use Delegating Constructors to Represent Common Actions for all Constructors of a Class». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-delegating.

параметров. Для этого можно использовать инициализатор элементов данных с именем класса:

```
Time::Time() : Time{0, 0, 0} {}
Time::Time(int hour) : Time{hour, 0, 0} {}
Time::Time(int hour, int minute) : Time{hour, minute, 0} {}
```

9.12. Деструкторы

Деструктор (destructor) — это специальная функция класса, у которой может не быть параметров или возвращаемого типа. Имя деструктора класса состоит из символа «тильда» (~) и последующего имени класса, например `~Time`. Это соглашение об именовании легко запоминается, потому что тильда — это оператор побитового дополнения, как вы увидите в следующей главе. В каком-то смысле деструктор является дополнением конструктора.

Деструктор класса неявно вызывается, когда объект уничтожается. Обычно это происходит, когда поток управления программы покидает ту область видимости, в которой был создан объект. Сам деструктор не удаляет объект из памяти. Он выполняет **завершающую обработку (termination housekeeping)**, например закрытие файла и очистку других используемых объектом ресурсов, прежде чем занимаемая объектом память будет освобождена для последующего использования.

Мы еще не определяли ни один деструктор в наших классах. Тем не менее в каждом классе C++ обязательно есть один (и только один) деструктор. Если программист не определил деструктор явно, компилятор создает для класса деструктор по умолчанию, вызывающий деструкторы любых элементов данных класса[1]. В главе 12 мы объясним, почему из деструкторов не должны выдаваться исключения.

9.13. Вызов конструкторов и деструкторов

Конструкторы неявно вызываются, когда объекты создаются, а деструкторы неявно вызываются, когда объекты выходят из области видимости. Порядок этих вызовов зависит от области видимости объектов. Обычно деструкторы вызываются в обратном порядке по сравнению с вызовами соответствующих конструкторов, но как вы увидите на илл. 9.13–9.15, глобальные и статические (`static`) объекты могут изменять порядок вызова деструкторов.

Конструкторы и деструкторы глобальных объектов

Конструкторы глобальных объектов, то есть объектов, определенных в глобальной области видимости (также называемой глобальным пространством имен), вызы-

[1] В главе 10 вы увидите, что деструктор по умолчанию также уничтожает объекты класса, созданные путем наследования.

ваются до выполнения любой другой функции в этом же файле исходного кода. Порядок выполнения конструкторов глобальных объектов между несколькими файлами не регламентируется, поэтому глобальные объекты в отдельных файлах не должны зависеть друг от друга. Когда функция `main` завершается инструкцией `return` или закрывающей фигурной скобкой, деструкторы таких объектов вызываются в обратном порядке по сравнению с вызовами соответствующих конструкторов. Для экстренного завершения программы в случае фатальной неустранимой ошибки часто используется функция `exit`, вызывающая деструкторы глобальных объектов[1]. Функция `abort`, в отличие от `exit`, прерывает программу немедленно, не допуская даже выполнения деструкторов или какого-либо другого завершающего кода. Как правило, вызов функции `abort` означает аварийное прекращение работы программы[2].

Конструкторы и деструкторы нестатических локальных объектов

Конструктор нестатического локального объекта вызывается, когда поток управления программы достигает определения этого объекта. Соответствующий деструктор объекта вызывается, когда поток управления программы выходит из области видимости объекта, то есть когда блок, в котором определен этот объект, завершается (нормально или из-за исключения). Деструкторы локальных объектов не вызываются, если программа завершается вызовом функции `exit` или `abort`.

Конструкторы и деструкторы статических локальных объектов

Конструктор статического (`static`) локального объекта вызывается только один раз, когда поток управления программы впервые достигает определения этого объекта. Соответствующий деструктор объекта вызывается, когда завершается функция `main` или вызывается функция `exit`. Глобальные и статические объекты уничтожаются в обратном порядке по сравнению с их созданием. Деструкторы статических объектов не вызываются, если программа завершается вызовом функции `abort`.

Демонстрация порядка вызова конструкторов и деструкторов

На илл. 9.13–9.15 мы показываем порядок вызова конструкторов и деструкторов для глобальных, локальных и локальных статических объектов класса `CreateAndDestroy` (илл. 9.13 и 9.14). Этот пример предназначен исключительно для учебных целей.

В коде на илл. 9.13 объявляется класс `CreateAndDestroy`. В строках 13–14 объявляются элементы данных класса: целое число (`m_ID`) и строка (`m_message`), которые идентифицируют каждый объект в выходных данных программы.

---

[1] «std::exit». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/utility/program/exit.

[2] «std::abort». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/utility/program/abort.

```
// Fig. 9.13: CreateAndDestroy.h
// Определение класса CreateAndDestroy
// Функции класса определены в CreateAndDestroy.cpp
#pragma once // Предотвращаем повторные включения заголовка
#include <string>
#include <string_view>

class CreateAndDestroy {
public:
   CreateAndDestroy(int ID, std::string_view message); // Конструктор
   ~CreateAndDestroy(); // Деструктор
private:
   int m_ID; // Идентификационный номер объекта
   std::string m_message; // Описание объекта
};
```

Илл. 9.13 | Определение класса CreateAndDestroy

Конструкторы и деструкторы (илл. 9.14) выводят на экран сообщения, чтобы показать, в каком порядке они вызываются. В деструкторе условное выражение (строка 18) проверяет, имеет ли уничтожаемый объект значение `m_ID`, равное 1 или 6. Если условие истинно, выводится символ новой строки для наглядности выходных данных.

```
// Fig. 9.14: CreateAndDestroy.cpp
// Определения функций класса CreateAndDestroy
#include <fmt/format.h>
#include <iostream>
#include "CreateAndDestroy.h" // Определение класса CreateAndDestroy

// Конструктор задает идентификационный номер и описание объекта
CreateAndDestroy::CreateAndDestroy(int ID, std::string_view message)
   : m_ID{ID}, m_message{message} {
   std::cout << fmt::format("Object {} constructor runs {}\n",
      m_ID, m_message);
}

// Деструктор
CreateAndDestroy::~CreateAndDestroy() {
   // Для наглядности выводим символ новой строки, если m_ID равно 1 или 6
   std::cout << fmt::format("{}Object {} destructor runs {}\n",
      (m_ID == 1 || m_ID == 6 ? "\n" : ""), m_ID, m_message);
}
```

Илл. 9.14 | Определения функций класса CreateAndDestroy

В коде на илл. 9.15 сначала определяется объект в глобальной области видимости (строка 9). Его конструктор вызывается перед выполнением любых инструкций в функции `main`, а его деструктор вызывается при завершении программы после деструкторов всех объектов с автоматической продолжительностью хранения.

```
// fig09_15.cpp
// Порядок вызова конструкторов
// и деструкторов
#include <iostream>
#include "CreateAndDestroy.h" // Определение класса CreateAndDestroy

void create(); // Прототип

const CreateAndDestroy first{1, "(global before main)"}; // Глобальный объект

int main() {
   std::cout << "\nMAIN FUNCTION: EXECUTION BEGINS\n";
   const CreateAndDestroy second{2, "(local in main)"};
   static const CreateAndDestroy third{3, "(local static in main)"};

   create(); // Вызываем функцию для создания объектов

   std::cout << "\nMAIN FUNCTION: EXECUTION RESUMES\n";
   const CreateAndDestroy fourth{4, "(local in main)"};
   std::cout << "\nMAIN FUNCTION: EXECUTION ENDS\n";
}

// Функция для создания объектов
void create() {
   std::cout << "\nCREATE FUNCTION: EXECUTION BEGINS\n";
   const CreateAndDestroy fifth{5, "(local in create)"};
   static const CreateAndDestroy sixth{6, "(local static in create)"};
   const CreateAndDestroy seventh{7, "(local in create)"};
   std::cout << "\nCREATE FUNCTION: EXECUTION ENDS\n";
}
```

```
Object 1 constructor runs (global before main)

MAIN FUNCTION: EXECUTION BEGINS
Object 2 constructor runs (local in main)
Object 3 constructor runs (local static in main)

CREATE FUNCTION: EXECUTION BEGINS
Object 5 constructor runs (local in create)
Object 6 constructor runs (local static in create)
Object 7 constructor runs (local in create)

CREATE FUNCTION: EXECUTION ENDS
Object 7 destructor runs (local in create)
Object 5 destructor runs (local in create)

MAIN FUNCTION: EXECUTION RESUMES
Object 4 constructor runs (local in main)

MAIN FUNCTION: EXECUTION ENDS
Object 4 destructor runs (local in main)
Object 2 destructor runs (local in main)
```

```
Object 6 destructor runs (local static in create)
Object 3 destructor runs (local static in main)

Object 1 destructor runs (global before main)
```

Илл. 9.15 | Порядок вызова конструкторов и деструкторов

Функция `main` (строки 11–21) определяет три объекта. Объекты `second` (строка 13) и `fourth` (строка 19) являются локальными, а объект `third` (строка 14) является статическим локальным. Конструктор каждого объекта вызывается, когда поток управления программы достигает точки, в которой этот объект определен. Когда поток управления программы достигает конца функции `main`, деструкторы объектов `fourth` и `second` вызываются в обратном порядке по сравнению с вызовами их конструкторов. Объект `third` является статическим, поэтому он существует до завершения программы. Деструктор объекта `third` вызывается перед деструктором глобального объекта `first`, но после уничтожения нестатических локальных объектов.

Функция `create` (строки 24–30) определяет три объекта: локальные автоматические объекты `fifth` (строка 26) и `seventh` (строка 28) и статический локальный объект `sixth` (строка 27). Когда функция `create` завершается, деструкторы объектов `seventh` и `fifth` вызываются в обратном порядке по сравнению с вызовами их конструкторов. Поскольку объект `sixth` является статическим, он существует до завершения программы. Деструктор объекта `sixth` вызывается перед деструкторами объектов `third` и `first`, но после уничтожения всех остальных нестатических объектов. В качестве упражнения добавьте еще один вызов функции `create` — вы увидите, что конструктор статического объекта `sixth` вызывается только один раз при первом вызове функции `create`.

9.14. Коварная ловушка: возврат ссылки или указателя на скрытый элемент данных

Ссылка на объект является псевдонимом для имени объекта, поэтому ее можно использовать в левой части инструкции присваивания. В этом контексте ссылка представляет собой вполне приемлемое значение *lvalue*, которое можно изменять.

Функция класса может возвращать ссылку на скрытый (`private`) элемент данных. Если возвращаемое значение является константной (`const`) ссылкой, как у функции `getName` нашего класса `Account` ранее в этой главе, эта ссылка является неизменяемым значением *lvalue* и не может быть использована для изменения данных. Но если возвращаемое значение является неконстантной ссылкой, могут возникнуть ошибки, которые трудно обнаружить.

На илл. 9.16–9.18 показан пример с упрощенным классом `Time`, демонстрирующий риск возврата ссылки на скрытый элемент данных. Функция `badSetHour` класса `Time` (объявленная на илл. 9.16 в строке 12 и определенная на илл. 9.17 в стро-

ках 30–33) возвращает ссылку `int&` на элемент данных `m_hour`. Такой возврат ссылки делает результат вызова функции `badSetHour` псевдонимом для скрытого элемента данных `hour`. Этот вызов функции можно использовать как скрытый элемент данных, в том числе как значение *lvalue* в инструкции присваивания. Таким образом, клиенты класса могут по своему желанию перезаписывать скрытые данные класса. Аналогичная проблема возникла бы, если бы функция возвращала указатель на скрытый элемент данных.

```
1 // Fig. 9.16: Time.h
2 // Определение класса Time
3 // Функции класса определены в Time.cpp
4
5 // Предотвращаем повторные включения заголовка
6 #pragma once
7
8 class Time {
9 public:
10    void setTime(int hour, int minute, int second);
11    int getHour() const;
12    int& badSetHour(int hour); // Опасный возврат ссылки!
13 private:
14    int m_hour{0};
15    int m_minute{0};
16    int m_second{0};
17 };
```

Илл. 9.16 | Определение класса Time

```
1 // Fig. 9.17: Time.cpp
2 // Определения функций класса Time
3 #include <stdexcept>
4 #include "Time.h" // Определение класса Time
5
6 // Установка нового значения времени в 24-часовом формате
7 void Time::setTime(int hour, int minute, int second) {
8    // Проверяем корректность hour, minute и second
9    if (hour < 0 || hour >= 24) {
10       throw std::invalid_argument{"hour was out of range"};
11    }
12
13    if (minute < 0 || minute >= 60) {
14       throw std::invalid_argument{"minute was out of range"};
15    }
16
17    if (second < 0 || second >= 60) {
18       throw std::invalid_argument{"second was out of range"};
19    }
20
21    m_hour = hour;
22    m_minute = minute;
```

```
23    m_second = second;
24 }
25
26 // Возвращаем значение hour
27 int Time::getHour() const {return m_hour;}
28
29 // Это неправильно: возврат ссылки на скрытый элемент данных!
30 int& Time::badSetHour(int hour) {
31    setTime(hour, m_minute, m_second);
32    return m_hour; // Опасный возврат ссылки!
33 }
```

Илл. 9.17 | Определения функций класса Time

В коде на илл. 9.18 объявляется объект `t` класса `Time` (строка 9) и ссылка `hourRef` (строка 12), которую мы инициализируем ссылкой, возвращенной в результате вызова `t.badSetHour(20)`. В строках 14–15 мы выводим на экран значение `hourRef`, демонстрируя нарушение инкапсуляции класса.

```
 1 // fig09_18.cpp
 2 // Открытая функция класса,
 3 // возвращающая ссылку на скрытые данные
 4 #include <iostream>
 5 #include <fmt/format.h>
 6 #include "Time.h" // Определение класса Time
 7
 8 int main() {
 9    Time t{}; // Создаем объект класса Time
10
11    // Инициализируем hourRef ссылкой, которую возвратила функция badSetHour
12    int& hourRef{t.badSetHour(20)}; // 20 - это допустимое значение
13
14    std::cout << fmt::format(
15       "Valid hour before modification: {}\n", hourRef);
16    hourRef = 30; // Присваиваем элементу недопустимое значение!
17    std::cout << fmt::format(
18       "Invalid hour after modification: {}\n\n", t.getHour());
19
20    // Опасно: вызов функции, возвращающей ссылку, может использоваться
21    // как lvalue! ПЛОХОЙ СТИЛЬ ПРОГРАММИРОВАНИЯ!!!
22    t.badSetHour(12) = 74; // Присваиваем элементу недопустимое значение!
23
24    std::cout << "After using t.badSetHour(12) as an lvalue, "
25       << fmt::format("hour is: {}\n", t.getHour());
26 }
```

```
Valid hour before modification: 20
Invalid hour after modification: 30
After using t.badSetHour(12) as an lvalue, hour is: 74
```

Илл. 9.18 | Открытая функция класса, возвращающая ссылку на скрытые данные

Никакие инструкции в функции `main` не должны иметь доступа к скрытым данным в объекте класса `Time`. Инструкция в строке 16 присваивает элементу данных `hour` недопустимое значение `30` через ссылку `hourRef`. В строках 17–18 мы вызываем функцию `getHour`, чтобы показать, что эта инструкция изменила скрытый элемент данных объекта `t`. В строке 22 вызов функции `badSetHour` используется в качестве значения *lvalue* и через возвращаемую этой функцией ссылку элементу данных `hour` присваивается недопустимое значение `74`. В строках 24–25 мы снова вызываем функцию `getHour`, чтобы показать, что инструкция в строке 22 изменила скрытый элемент данных объекта `t`.

Возврат ссылки или указателя на скрытый (`private`) элемент данных нарушает инкапсуляцию класса, делая клиентский код зависимым от представления данных в классе. Лишь в некоторых случаях это может быть уместно. Мы рассмотрим один из таких случаев, когда напишем класс `myArray` в разделе 11.6.

9.15. Оператор присваивания по умолчанию

Оператор присваивания (`=`) может присвоить объект другому объекту того же типа. **Оператор присваивания по умолчанию (default assignment operator)**[1], сгенерированный компилятором, копирует каждый элемент данных из правого операнда в соответствующий элемент данных в левом операнде. На илл. 9.19 и 9.20 определяется класс `Date`. В строке 15 на илл. 9.21 объект `date1` класса `Date` присваивается объекту `date2` класса `Date`. Это означает, что элементы `m_year`, `m_month` и `m_day` объекта `date2` получают значения `m_year`, `m_month` и `m_day` соответственно, взятые из объекта `date1`.

```
1 // Fig. 9.19: Date.h
2 // Определение класса Date. Функции класса определены в Date.cpp
3 #pragma once // Предотвращаем повторные включения заголовка
4 #include <string>
5
6 // Определение класса Date
7 class Date {
8 public:
9    Date(int year, int month, int day);
10   std::string toString() const;
11 private:
12   int m_year;
13   int m_month;
14   int m_day;
15 };
```

Илл. 9.19 | Определение класса Date

[1] Точнее, это оператор копирующего присваивания по умолчанию. В главе 11 мы рассмотрим различие между оператором копирующего присваивания и оператором перемещающего присваивания.

```
// Fig. 9.20: Date.cpp
// Определения функций класса Date
#include <fmt/format.h>
#include <string>
#include "Date.h" // Определение класса Date

// Конструктор класса Date (должен проверять допустимость значений)
Date::Date(int year, int month, int day)
   : m_year{year}, m_month{month}, m_day{day} {}

// Возвращаем строку с датой в формате yyyy-mm-dd
std::string Date::toString() const {
   return fmt::format("{}-{:02d}-{:02d}", m_year, m_month, m_day);
}
```

Илл. 9.20 | Определения функций класса Date

```
// fig09_21.cpp
// Присваивание одного объекта другому
// с помощью оператора присваивания по умолчанию
#include <fmt/format.h>
#include <iostream>
#include "Date.h" // Определение класса Date
using namespace std;

int main() {
   const Date date1{2006, 7, 4};
   Date date2{2022, 1, 1};

   std::cout << fmt::format("date1: {}\ndate2: {}\n\n",
      date1.toString(), date2.toString());
   date2 = date1; // Используем оператор присваивания по умолчанию
   std::cout << fmt::format("After assignment, date2: {}\n",
      date2.toString());
}
```

```
date1: 2006-07-04
date2: 2022-01-01

After assignment, date2: 2006-07-04
```

Илл. 9.21 | Присваивание одного объекта другому с помощью оператора присваивания по умолчанию

Копирующие конструкторы

Функции могут получать объекты в качестве аргументов, а также возвращать объекты. По умолчанию каждый объект при этом передается по значению, то есть функция передает или возвращает копию объекта. В таких случаях C++ создает новый объект и вызывает **копирующий конструктор (copy constructor)**, чтобы скопировать все данные из исходного объекта в новый. Для всех классов,

которые мы до сих пор рассматривали, компилятор предоставляет копирующий конструктор по умолчанию, который копирует каждый элемент из исходного объекта в соответствующий элемент нового объекта[1].

9.16. Константные объекты и константные функции класса

Рассмотрим применение принципа наименьших привилегий к объектам. Некоторые объекты никогда не должны изменяться, поэтому их надо объявлять как константные (с квалификатором `const`). Любая попытка изменить константный объект приводит к ошибке компиляции. Инструкция

```
const Time noon{12, 0, 0};
```

объявляет константный объект `noon` класса `Time` и инициализирует его значениями, соответствующими полудню (12 PM). Вы можете создавать константные и неконстантные объекты одного и того же класса.

C++ запрещает вызов функции константного объекта, если она сама не является константной. Поэтому объявляйте с квалификатором `const` каждую функцию, не изменяющую объект, для которого она вызывается.

Конструктор должен изменять объект в процессе инициализации. Деструктор должен выполнять завершающую обработку до того, как память объекта будет освобождена и передана операционной системе. Поэтому объявление конструктора или деструктора с квалификатором `const` приведет к ошибке компиляции. Неизменяемость каждого константного объекта обеспечивается с момента его создания на протяжении всего времени его существования.

Использование константных и неконстантных функций класса

Наша следующая программа (илл. 9.22) использует почти такой же класс `Time`, как на илл. 9.10 и 9.11, но с одним отличием: мы убрали квалификатор `const` в прототипе и в определении функции `to12HourString`, чтобы вызвать ошибку компиляции. Мы создаем два объекта класса `Time`: неконстантный объект `wakeUp` (строка 6) и константный объект `noon` (строка 7). Программа пытается вызвать неконстантные функции для объекта `noon`: `setHour` (строка 11) и `to12HourString` (строка 15). В обоих случаях компилятор выдает сообщение об ошибке. Программа также иллюстрирует три другие комбинации объектов и функций:

- вызов неконстантной функции для неконстантного объекта (строка 10);
- вызов константной функции для неконстантного объекта (строка 12);
- вызов константной функции для константного объекта (строки 13–14).

[1] В главе 11 мы рассмотрим те случаи, когда компилятор заменяет копирующий конструктор перемещающим конструктором.

После кода программы показаны сообщения об ошибках, возникающих при вызове неконстантной функции для константного объекта. Мы добавили пустые строки для наглядности.

```
1 // fig09_22.cpp
2 // Константные объекты и константные функции класса
3 #include "Time.h" // Определение класса Time
4
5 int main() {
6    Time wakeUp{6, 45, 0}; // Неконстантный объект
7    const Time noon{12, 0, 0}; // Константный объект
8
9                              // Сочетания объектов и функций
10   wakeUp.setHour(18);       // Неконстантный объект, неконстантная функция
11   noon.setHour(12);         // Константный объект, неконстантная функция!!!
12   wakeUp.getHour();         // Неконстантный объект, константная функция
13   noon.getMinute();         // Константный объект, константная функция
14   noon.to24HourString();    // Константный объект, константная функция
15   noon.to12HourString();    // Константный объект, неконстантная функция!!!
16 }
```

Сообщения компилятора clang++ об ошибках:

```
fig09_22.cpp:11:4: error: 'this' argument to member function 'setHour' has type
'const Time', but function is not marked const
   noon.setHour(12); // const non-const
   ^~~~
./Time.h:15:9: note: 'setHour' declared here
   void setHour(int hour); // set hour (after validation)
        ^
fig09_22.cpp:15:4: error: 'this' argument to member function 'to12HourString'
has type 'const Time', but function is not marked const
   noon.to12HourString(); // const non-const
   ^~~~
./Time.h:25:16: note: 'to12HourString' declared here
   std::string to12HourString(); // 12-hour time format string
               ^
2 errors generated.
```

Илл. 9.22 | Константные объекты и константные функции класса

Конструктор должен быть неконстантной функцией класса, но он может инициализировать константный объект (илл. 9.22, строка 7). Напомним, что в определении конструктора класса `Time` (илл. 9.11) вызывается неконстантная функция `setTime` для инициализации объекта класса `Time`. Вызов неконстантной функции конструктора для константного объекта разрешен. Объект не будет константным, пока конструктор не завершит его инициализацию.

В строке 15 на илл. 9.22 возникает ошибка компиляции, несмотря на то что функция `to12HourString` класса `Time` не изменяет объект, из которого она вызывается.

Этого факта недостаточно: вы должны явно объявить функцию с квалификатором `const`, чтобы такой вызов был разрешен компилятором.

9.17. Композиция: объекты как элементы классов

Предположим, что объект `AlarmClock` должен знать, в какое время надо подать сигнал будильника. Почему бы не включить в этот класс объект `Time` в качестве элемента? Такая возможность повторного использования кода называется **композицией (composition)**, или **агрегацией (aggregation)**. Элементами класса могут быть объекты других классов[1]. Вы использовали композицию в примерах этой главы с классом `Account`, который содержал объект класса `string` в качестве элемента данных.

Вы уже знаете, как передавать аргументы конструктору. Теперь рассмотрим, как конструктор класса передает аргументы конструкторам объектов-элементов класса, используя инициализаторы. В следующей программе с классами `Date` (илл. 9.23 и 9.24) и `Employee` (илл. 9.25 и 9.26) мы покажем пример композиции. Класс `Employee` (илл. 9.25) содержит скрытые элементы данных `m_firstName`, `m_lastName`, `m_birthDate` и `m_hireDate`. Элементы этого класса `m_birthDate` и `m_hireDate` являются объектами класса `Date`, который содержит скрытые элементы данных `m_year`, `m_month` и `m_day`.

```
 1 // Fig. 9.23: Date.h
 2 // Определение класса Date; функции класса определены в Date.cpp
 3 #pragma once // Предотвращаем повторные включения заголовка
 4 #include <string>
 5
 6 class Date {
 7 public:
 8    static const int monthsPerYear{12}; // Месяцы в году
 9    Date(int year, int month, int day);
10    std::string toString() const; // Дата в формате yyyy-mm-dd
11    ~Date(); // Реализация показывает, когда вызывается деструктор
12 private:
13    int m_year; // Любой год
14    int m_month; // 1-12 (месяц)
15    int m_day; // 1-31 в зависимости от месяца
16
17    // Служебная функция, проверяющая соответствие дня, месяца и года
18    bool checkDay(int day) const;
19 };
```

Илл. 9.23 | Определение класса Date

[1] Как вы увидите в главе 10, при создании класса можно скопировать его атрибуты и поведение из другого класса — это называется наследованием.

```
// Fig. 9.24: Date.cpp
// Определения функций класса Date
#include <array>
#include <fmt/format.h>
#include <iostream>
#include <stdexcept>
#include "Date.h" // Определение класса Date

// Конструктор проверяет допустимость значения month и вызывает
// служебную функцию checkDay для проверки допустимости значения day
Date::Date(int year, int month, int day)
   : m_year{year}, m_month{month}, m_day{day} {
   if (m_month < 1 || m_month > monthsPerYear) { // Проверяем значение month
      throw std::invalid_argument{"month must be 1-12"};
   }

   if (!checkDay(day)) { // Проверяем значение day
      throw std::invalid_argument{
         "Invalid day for current month and year"};
   }

   // Сообщение показывает момент вызова конструктора объекта Date
   std::cout << fmt::format("Date object constructor: {}\n", toString());
}

// Получаем дату в формате yyyy-mm-dd
std::string Date::toString() const {
   return fmt::format("{}-{:02d}-{:02d}", m_year, m_month, m_day);
}

// Сообщение показывает момент вызова деструктора объекта Date
Date::~Date() {
   std::cout << fmt::format("Date object destructor: {}\n", toString());
}

// Служебная функция, проверяющая допустимость значения day
// для заданных month и year (учитывает високосные годы)
bool Date::checkDay(int day) const {
   // Элемент 0 игнорируется
   static const std::array daysPerMonth{
      0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

   // Проверяем допустимость значения day для заданного month
   if (1 <= day && day <= daysPerMonth.at(m_month)) {
      return true;
   }

   // Проверка на високосный год (29 февраля)
   if (m_month == 2 && day == 29 && (m_year % 400 == 0 ||
      (m_year % 4 == 0 && m_year % 100 != 0))) {
      return true;
```

Окончание ⇨

```
52    }
53
54    return false; // Недопустимое значение day для заданных m_month и m_year
55 }
```

Илл. 9.24 | Определения функций класса Date

```
 1 // Fig. 9.25: Employee.h
 2 // Определение класса Employee, демонстрирующего композицию
 3 // Функции класса определены в Employee.cpp
 4 #pragma once // Предотвращаем повторные включения заголовка
 5 #include <string>
 6 #include <string_view>
 7 #include "Date.h" // Определение класса Date
 8
 9 class Employee {
10 public:
11    Employee(std::string_view firstName, std::string_view lastName,
12       const Date& birthDate, const Date& hireDate);
13    std::string toString() const;
14    ~Employee(); // Для демонстрации порядка выполнения деструкторов
15 private:
16    std::string m_firstName; // Композиция: объект-элемент
17    std::string m_lastName; // Композиция: объект-элемент
18    Date m_birthDate; // Композиция: объект-элемент
19    Date m_hireDate; // Композиция: объект-элемент
20 };
```

Илл. 9.25 | Определение класса Employee, демонстрирующего композицию

```
 1 // Fig. 9.26: Employee.cpp
 2 // Определения функций класса Employee
 3 #include <fmt/format.h>
 4 #include <iostream>
 5 #include "Employee.h" // Определение класса Employee
 6 using namespace std;
 7
 8 // Конструктор использует список инициализаторов для передачи значений
 9 // инициализаторов конструкторам объектов-элементов
10 Employee::Employee(std::string_view firstName, std::string_view lastName,
11    const Date &birthDate, const Date &hireDate)
12    : m_firstName{firstName}, m_lastName{lastName},
13      m_birthDate{birthDate}, m_hireDate{hireDate} {
14    // Сообщение показывает момент вызова конструктора объекта Employee
15    std::cout << fmt::format("Employee object constructor: {} {}\n",
16       m_firstName, m_lastName);
17 }
18
19 // Получаем строковое представление объекта Employee
20 std::string Employee::toString() const {
```

```
21    return fmt::format("{}, {} Hired: {} Birthday: {}", m_lastName,
22       m_firstName, m_hireDate.toString(), m_birthDate.toString());
23 }
24
25 // Сообщение показывает момент вызова деструктора объекта Employee
26 Employee::~Employee() {
27    cout << fmt::format("Employee object destructor: {}, {}\n",
28       m_lastName, m_firstName);
29 }
```

Илл. 9.26 | Определения функций класса Employee

Список инициализаторов конструктора класса Employee

Согласно прототипу конструктора класса `Employee` (илл. 9.25, строки 11–12), конструктор имеет четыре параметра: `firstName`, `lastName`, `birthDate` и `hireDate`. В определении этого конструктора (илл. 9.26, строки 12–13) первые два параметра передаются через инициализаторы элементов в конструктор класса `string` для элементов данных `FirstName` и `LastName`. Последние два параметра передаются через список инициализаторов в конструктор класса `Date` для элементов данных `birthDate` и `HireDate`. Порядок перечисления инициализаторов в списке не имеет значения. Элементы данных создаются в том порядке, в каком они объявлены в классе `Employee`. В C++ Core Guidelines рекомендуется перечислять инициализаторы элементов в том же порядке, в каком они объявлены в классе[1].

Копирующий конструктор класса Date по умолчанию

Изучая код класса `Date` (илл. 9.23), обратите внимание, что он не предоставляет конструктор с параметром `Date`. Почему же инициализатор в конструкторе `Employee` может инициализировать объекты `m_birthDate` и `m_hireDate`, передавая их конструкторам объекты класса `Date`? Напомним из раздела 9.15, что компилятор предоставляет каждому классу копирующий конструктор по умолчанию, который копирует каждый элемент данных из аргумента конструктора в соответствующий элемент данных инициализируемого объекта. В главе 11 вы узнаете, как определять свои собственные копирующие конструкторы.

Тестирование классов Date и Employee

В программе, показанной на илл. 9.27, мы создаем два объекта `Date` (строки 9–10) и передаем их в качестве аргументов конструктору объекта `Employee`, созданного в строке 11. Когда создается объект `Employee`, его конструктор выполняет:

- два вызова конструктора класса `string` (строка 12 на илл. 9.26);
- два вызова копирующего конструктора класса `Date` по умолчанию (строка 13 на илл. 9.26).

[1] C++ Core Guidelines, «C.47: Define and Initialize Member Variables in the Order of Member Declaration». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-order.

В строке 13 на илл. 9.27 мы выводим на экран данные объекта **Employee**, подтверждающие корректность инициализации.

```
1 // fig09_27.cpp
2 // Пример композиции: объект, содержащий объекты-элементы
3 #include <fmt/format.h>
4 #include <iostream>
5 #include "Date.h" // Определение класса Date
6 #include "Employee.h" // Определение класса Employee
7
8 int main() {
9    const Date birth{1987 ,7, 24};
10   const Date hire{2018, 3, 12};
11   const Employee manager{"Sue", "Green", birth, hire};
12
13   std::cout << fmt::format("\n{}\n\n", manager.toString());
14 }
```

```
Date object constructor: 1987-07-24
Date object constructor: 2018-03-12
Employee object constructor: Sue Green

Green, Sue Hired: 2018-03-12 Birthday: 1987-07-24

Employee object destructor: Green, Sue
Date object destructor: 2018-03-12
Date object destructor: 1987-07-24
Date object destructor: 2018-03-12
Date object destructor: 1987-07-24
```

Илл. 9.27 | Пример композиции: объект, содержащий объекты-элементы

Когда создаются объекты класса **Date** (строки 9–10), вызываемый конструктор класса **Date** (строки 11–24 на илл. 9.24) выводит на экран сообщения, демонстрирующие эти вызовы (см. первые две строки выходных данных). Однако в строке 11 на илл. 9.27 происходят два вызова копирующего конструктора класса **Date** (строка 13 на илл. 9.26), которые не отображаются в выходных данных этой программы. Дело в том, что копирующий конструктор класса **Date** создается компилятором и не содержит никаких инструкций вывода на экран.

Каждый из классов **Date** и **Employee** содержит деструктор (строки 32–34 на илл. 9.24 и строки 26–29 на илл. 9.26 соответственно), который выводит сообщение при уничтожении объекта этого класса. Деструкторы помогают нам показать порядок уничтожения объектов: объекты класса **Date** уничтожаются после объекта класса **Employee**.

Обратите внимание на последние четыре строки выходных данных на илл. 9.27. Последние две строки — это сообщения от деструктора класса **Date**, работающего с объектами **hire** (илл. 9.27, строка 10) и **birth** (строка 9) соответственно. Выходные данные подтверждают, что все три объекта, созданные в функции **main**,

уничтожаются в обратном порядке по сравнению с их созданием. Сообщение от деструктора объекта `Employee` находится на пятой строке снизу. Четвертая и третья строки снизу — это сообщения от деструкторов объектов класса `Employee`: `m_hireDate` (илл. 9.25, строка 19) и `m_birthDate` (строка 18).

Эти выходные данные подтверждают уничтожение объекта `Employee` в порядке «снаружи внутрь». Деструктор объекта `Employee` запускается первым (см. пятую снизу строку в выходных данных). Затем объекты-элементы уничтожаются в обратном порядке по сравнению с их созданием. Деструктор класса `string` не содержит инструкций вывода, поэтому мы не видим процесс уничтожения объектов `firstName` и `LastName`.

Что произойдет, если не использовать список инициализаторов?

Если вы не инициализировали объект-элемент явно, то для инициализации неявно вызывается его конструктор по умолчанию. При отсутствии конструктора по умолчанию возникает ошибка компиляции. Значения, установленные конструктором по умолчанию, можно впоследствии изменить с помощью *set*-функций, но при таком подходе компьютер будет делать дополнительную работу и инициализация займет больше времени. Рекомендуется сразу инициализировать объекты-элементы инициализаторами, устраняя тем самым лишнюю нагрузку на компьютер, вызванную их повторной инициализацией.

9.18. Дружественные функции и классы

Дружественной функцией (friend function) по отношению к какому-либо классу называют такую функцию, которая имеет доступ к открытым и скрытым элементам этого класса. Друзьями класса могут быть:

- автономные функции,
- классы (и, следовательно, все их функции);
- некоторые функции других классов.

В этом разделе мы покажем вам простой пример дружественной функции. В главе 11 мы рассмотрим дружественные функции, которые перегружают операторы для работы с объектами пользовательских классов. Вы увидите, что иногда функцию класса нельзя использовать для определения некоторых перегруженных операторов.

Объявление дружественных отношений

Чтобы объявить дружественную классу функцию, поместите в определение этого класса ее прототип с предшествующим ключевым словом `friend`. Чтобы объявить все функции существующего класса `classTwo` дружественными классу `ClassOne`, поместите в определение класса `ClassOne` такое объявление:

```
friend class ClassTwo;
```

Правила дружбы

Основные правила:

- Дружба предоставляется, а не забирается. Чтобы класс B стал другом класса A, класс A должен объявить класс B своим другом.
- Дружба не симметрична. Если класс A является другом класса B, из этого не следует, что класс B является другом класса A.
- Дружба не передается. Если класс A является другом класса B, а класс B является другом класса C, из этого не следует, что класс A является другом класса C.

Игнорирование спецификаторов доступа

Спецификаторы доступа `public`, `protected` (глава 10) и `private` не применяются к объявлениям друзей класса, поэтому такие объявления можно размещать в любом месте определения класса. Мы предпочитаем размещать объявления друзей класса в начале определения класса, без какого-либо спецификатора доступа перед ними.

Изменение скрытых данных класса дружественной функцией

В следующей программе (илл. 9.28) функция `modifyX` определяется как дружественная к классу `Count` (строка 9). Таким образом, функция `modifyX` может изменять скрытый элемент данных `m_x`.

```
 1 // fig09_28.cpp
 2 // Дружественная функция получает доступ к скрытым элементам данных класса
 3 #include <fmt/format.h>
 4 #include <iostream>
 5 #include "fmt/format.h" // Или в C++20: #include <format>
 6
 7 // Определение класса Count
 8 class Count {
 9    friend void setX(Count& c, int value); // Объявляем дружественную функцию
10 public:
11    int getX() const {return m_x;}
12 private:
13    int m_x{0};
14 };
15
16 // Функция modifyX может изменять скрытые данные класса Count,
17 // поскольку она объявлена как дружественная к этому классу (строка 8)
18 void modifyX(Count& c, int value) {
19    c.m_x = value; // Допустимо, так как modifyX - это друг класса Count
20 }
21
22 int main() {
23    Count counter{}; // Создаем объект класса Count
24
```

```
25    std::cout << fmt::format("Initial counter.m_x: {}\n", counter.getX());
26    modifyX(counter, 8); // Изменяем значение скрытого элемента
27    std::cout << fmt::format("counter.m_x after modifyX: {}\n",
28       counter.getX());
29 }
```

```
Initial counter.m_x: 0
counter.m_x after modifyX: 8
```

Илл. 9.28 | Дружественная функция получает доступ к скрытым элементам данных класса

Функция `modifyX` (строки 18–20) — это автономная (свободная) функция, не входящая в класс `Count`. Когда мы вызываем из `main` функцию `modifyX` для изменения объекта `counter` класса `Count` (строка 26), мы должны передать ей этот объект в качестве аргумента. Функция `modifyX` получает доступ к скрытому элементу данных `m_x` (строка 19), потому что она объявлена как дружественная классу `Count` (строка 9). Без этого объявления мы получили бы сообщения об ошибках, указывающие, что у функции `modifyX` нет доступа к скрытому элементу данных `m_x`.

9.19. Указатель this

Во время работы программы набор функций каждого класса существует в памяти компьютера только в одном экземпляре. Тем не менее объектов каждого класса может быть сколько угодно. Каким же образом вызванная функция находит среди многих объектов именно тот, с элементами данных которого ей надо работать? Она обращается к объекту через указатель `this` (ключевое слово C++), инициализированный неявным аргументом: адресом конкретного объекта. Он передается всем нестатическим[1] функциям объекта.

Использование указателя this для предотвращения конфликтов имен

Функции класса используют указатель `this` неявно (как мы делали до сих пор) или явно, чтобы обращаться к элементам данных и к другим функциям объекта. Одно из явных применений указателя `this` заключается в предотвращении конфликтов имен между элементами данных класса и параметрами конструктора или функций класса. Если в функции есть локальная переменная и элемент данных с одинаковым именем, локальная переменная **скрывает** (**hide**), или **затеняет** (**shadow**), собой элемент данных. Использование в теле функции этого имени означает обращение к локальной переменной, а не к элементу данных.

В таких случаях для доступа к элементу данных надо квалифицировать его имя с помощью `this->`. Например, если в классе `Account` есть элемент с именем `name`, к нему можно обратиться из функции `setName`:

[1] В разделе 9.20 мы рассмотрим статические элементы класса и объясним, почему не происходит неявной передачи указателя this статическим функциям класса.

```
void Account::setName(std::string_view name) {
   this->name = name; // this-> служит для доступа к элементу данных объекта
}
```

Здесь `this->name` — это элемент данных с именем `name`. Впрочем, для предотвращения конфликтов имен рекомендуется начинать все имена элементов данных (в отличие от локальных переменных) с префикса `m_`, как мы делали во всех пользовательских классах.

Типы указателей this

Тип указателя `this` зависит от типа объекта и от того, является ли константной (`const`) та функция, в которой он используется:

- В неконстантной функции класса `Time` указатель `this` имеет тип `Time*` — указатель на объект класса `Time`.
- В константной функции класса `Time` указатель `this` имеет тип `const Time*` — константный указатель на объект класса `Time`.

9.19.1. Неявное и явное использование указателя this для доступа к элементам данных объекта

На илл. 9.29 показан пример, демонстрирующий неявное и явное использование указателя `this` в функции класса для отображения скрытого элемента данных `m_x` объекта `Test`. В разделе 9.19.2 и в главе 11 мы покажем еще несколько полезных примеров кода с указателем `this`.

```
// fig09_29.cpp
// Использование указателя this для обращения к элементам данных объекта
#include <fmt/format.h>
#include <iostream>

class Test {
public:
   explicit Test(int value);
   void print() const;
private:
   int m_x{0};
};

// Конструктор
Test::Test(int value) : m_x{value} {} // Инициализируем m_x значением value

// Отображаем m_x, используя указатель this неявно, а затем явно
// Круглые скобки вокруг *this нужны для согласования приоритетов операторов
void Test::print() const {
   // Неявно используем указатель this для доступа к элементу m_x
   std::cout << fmt::format(" m_x = {}\n", m_x);
```

```
22
23    // Явно используем указатель this и оператор "стрелка"
24    // для доступа к элементу m_x
25    std::cout << fmt::format(" this->m_x = {}\n", this->m_x);
26
27    // Явно используем разыменованный указатель this
28    // и оператор "точка" для доступа к элементу m_x
29    std::cout << fmt::format("(*this).m_x = {}\n", (*this).m_x);
30 }
31
32 int main() {
33    const Test testObject{12}; // Создаем и инициализируем объект testObject
34    testObject.print();
35 }
```

```
        x = 12
  this->x = 12
(*this).x = 12
```

Илл. 9.29 | Использование указателя this для обращения к элементам данных объекта

Функция `print` (строки 19–30) сначала отображает значение `m_x`, неявно используя указатель `this` (строка 21) — указывается только имя элемента данных. Затем в функции `print` используются разные формы записи для доступа к `m_x` через указатель `this`:

- `this->m_x` (строка 25);
- `(*this).m_x` (строка 29).

Круглые скобки вокруг `*this` (строка 29) обязательны, поскольку оператор «точка» (`.`) имеет более высокий приоритет, чем оператор разыменования указателя (`*`). Без круглых скобок выражение `*this.m_x` было бы вычислено как `*(this.m_x)`. Оператор «точка» нельзя использовать с указателем, так как это вызовет ошибку компиляции.

9.19.2. Использование указателя this для каскадного вызова функций

Указатель `this` также применяют для **каскадного вызова функций класса (cascaded member-function calls)**, то есть для последовательного вызова нескольких функций в одной инструкции, как в строке 11 на илл. 9.32. Программа, показанная на илл. 9.30–9.32, содержит измененные функции класса `Time`: `setTime`, `setHour`, `setMinute` и `setSecond`. Каждая из них возвращает ссылку на объект класса `Time`, для которого вызывается. Эта ссылка позволяет выполнять каскадный вызов функций. На илл. 9.31 последняя инструкция в функции `setTime` возвращает ссылку на `*this` (строка 30). Каждая из функций `setHour`, `setMinute` и `setSecond` вызывает `setTime` и возвращает `Time&`.

```
// Fig. 9.30: Time.h
// Класс Time, измененный для каскадного вызова функций
#pragma once // Предотвращаем повторные включения заголовка
#include <string>

class Time {
public:
   // Конструктор по умолчанию, так как он может быть вызван без аргументов
   explicit Time(int hour = 0, int minute = 0, int second = 0);

   // Set-функции
   Time& setTime(int hour, int minute, int second);
   Time& setHour(int hour); // Устанавливаем hour (после проверки)
   Time& setMinute(int minute); // Устанавливаем minute (после проверки)
   Time& setSecond(int second); // Устанавливаем second (после проверки)

   int getHour() const; // Возвращаем hour
   int getMinute() const; // Возвращаем minute
   int getSecond() const; // Возвращаем second
   std::string to24HourString() const; // 24-часовой формат времени
   std::string to12HourString() const; // 12-часовой формат времени
private:
   int m_hour{0}; // 0 - 23 (24-часовой формат часов)
   int m_minute{0}; // 0 - 59
   int m_second{0}; // 0 - 59
};
```

Илл. 9.30 | Класс Time, измененный для каскадного вызова функций

```
// Fig. 9.31: Time.cpp
// Определения функций класса Time
#include <fmt/format.h>
#include <stdexcept>
#include "Time.h" // Определение класса Time

// Конструктор инициализирует каждый элемент данных
Time::Time(int hour, int minute, int second) {
   setTime(hour, minute, second);
}

// Устанавливаем новое время в 24-часовом формате
Time& Time::setTime(int hour, int minute, int second) {
   // Проверяем допустимость значений hour, minute и second
   if (hour < 0 || hour >= 24) {
      throw std::invalid_argument{"hour was out of range"};
   }

   if (minute < 0 || minute >= 60) {
      throw std::invalid_argument{"minute was out of range"};
```

```
21     }
22
23     if (second < 0 || second >= 60) {
24        throw std::invalid_argument("second was out of range");
25     }
26
27     m_hour = hour;
28     m_minute = minute;
29     m_second = second;
30     return *this; // Для каскадного вызова функций
31 }
32
33 // Устанавливаем значение hour
34 Time& Time::setHour(int hour) {
35     return setTime(hour, m_minute, m_second);
36 }
37
38 // Устанавливаем значение minute
39 Time& Time::setMinute(int minute) {
40     return setTime(m_hour, minute, m_second);
41 }
42
43 // Устанавливаем значение second
44 Time& Time::setSecond(int second) {
45     return setTime(m_hour, m_minute, second);
46 }
47
48 // Получаем значение hour
49 int Time::getHour() const {return m_hour;}
50
51 // Получаем значение minute
52 int Time::getMinute() const {return m_minute;}
53
54 // Получаем значение second
55 int Time::getSecond() const {return m_second;}
56
57 // Возвращаем время в 24-часовом формате (HH:MM:SS)
58 std::string Time::to24HourString() const {
59     return fmt::format("{:02d}:{:02d}:{:02d}",
60        getHour(), getMinute(), getSecond());
61 }
62
63 // Возвращаем время в 12-часовом формате (HH:MM:SS AM или PM)
64 std::string Time::to12HourString() const {
65     return fmt::format("{}:{:02d}:{:02d} {}",
66        ((getHour() % 12 == 0) ? 12 : getHour() % 12),
67        getMinute(), getSecond(), (getHour() < 12 ? "AM" : "PM"));
68 }
```

Илл. 9.31 | Определения функций класса Time, измененного для каскадного вызова функций

На илл. 9.32 мы создаем объект `t` класса `Time` (строка 9), а затем используем его в каскадном вызове функций (строки 11 и 19).

```
1 // fig09_32.cpp
2 // Каскадный вызов функций через указатель this
3 #include <fmt/format.h>
4 #include <iostream>
5 #include "Time.h" // Определение класса Time
6 using namespace std;
7
8 int main() {
9    Time t{}; // Создаем объект класса Time
10
11    t.setHour(18).setMinute(30).setSecond(22); // Каскадный вызов функций
12
13    // Отображаем время в 24-часовом и 12-часовом формате
14    std::cout << fmt::format("24-hour time: {}\n12-hour time: {}\n\n",
15       t.to24HourString(), t.to12HourString());
16
17    // Каскадный вызов функций
18    std::cout << fmt::format("New 12-hour time: {}\n",
19       t.setTime(20, 20, 20).to12HourString());
20 }
```

```
24-hour time: 18:30:22
12-hour time: 6:30:22 PM

New 12-hour time: 8:20:20 PM
```

Илл. 9.32 | Каскадный вызов функций через указатель this

Почему работает возврат `*this` в качестве ссылки? Оператор «точка» (`.`) группируется слева направо, поэтому инструкция в строке 11

```
t.setHour(18).setMinute(30).setSecond(22);
```

прежде всего вызывает функцию `t.setHour(18)`, которая возвращает ссылку на объект `t` (уже измененный) в качестве значения этого вызова функции. Оставшееся выражение интерпретируется так:

```
t.setMinute(30).setSecond(22);
```

Теперь вызывается функция `t.setMinute(30)`, которая возвращает ссылку на объект `t` (измененный повторно). Оставшееся выражение интерпретируется так:

```
t.setSecond(22);
```

В строке 19 на илл. 9.32 тоже используется каскадный вызов[1]. Мы не можем добавить в каскад другую функцию класса `Time` после вызова функции `to12HourString`,

[1] Этот конкретный каскадный вызов предназначен только для учебных целей. Вызовы функций, изменяющие объекты, надо помещать в отдельные инструкции, чтобы избежать проблем с порядком выполнения.

потому что она не возвращает `Time&`. Однако мы могли бы добавить функцию класса `string`, потому что `to12HourString` возвращает объект `string`. В главе 11 представлены практические примеры каскадных вызовов функций, например, в инструкциях `cout` с оператором << и в инструкциях `cin` с оператором >>.

9.20. Статические элементы класса: данные и функции, общие для всех объектов класса

Есть исключение из правила, согласно которому каждый объект имеет собственную копию элементов данных. Иногда требуется, чтобы некоторые переменные были общими для всех объектов класса. Для этой цели, а также по некоторым другим причинам используются **статические (static)** элементы данных (объявленные с ключевым словом `static`). Статическая переменная «работает на весь класс», то есть содержит данные, доступные каждому объекту класса. Статические элементы данных позволяют экономить память в тех случаях, когда все объекты класса могут совместно использовать единственный экземпляр данных.

Преимущества статических данных

Рассмотрим необходимость статических данных на простом примере. Предположим, что мы пишем игру с марсианами (объектами класса `Martian`) и другими космическими существами. Каждый `Martian` агрессивен и готов атаковать других существ, но только когда он уверен, что в данный момент на игровом поле действуют не менее пяти объектов `Martian`. А если их меньше пяти, каждый `Martian` становится трусливым. Таким образом, количество действующих марсиан (значение переменной `martianCount`) должно быть известно каждому из них. Предположим, что мы включим элемент данных `martianCount` во все объекты класса `Martian`. В этом случае у каждого марсианина будет собственная копия этой переменной. При создании и уничтожении каждого объекта `Martian` нам придется обновлять `martianCount` во всех объектах этого класса. Каждый объект `Martian` должен будет хранить в себе постоянно меняющееся количество объектов `Martian`. Из-за этого память компьютера будет занята бесполезными копиями `martianCount`, а для их синхронизации потребуется много процессорного времени. Вместо этого мы объявим элемент данных `martianCount` статическим (`static`), чтобы он стал общим для всех объектов класса `Martian`. Тогда каждый `Martian` получит доступ к `martianCount` как к собственному элементу данных, но в программе будет только один экземпляр `martianCount`. Это сэкономит память. Мы включим в конструктор класса `Martian` инструкцию инкремента статической переменной `martianCount`, а в деструктор — инструкцию ее декремента. Это гораздо проще, чем выполнять в процессе игры многократные инструкции инкремента и декремента для множества объектов.

Инициализация и область видимости статических элементов данных

Статические элементы данных имеют область видимости класса. Статический элемент данных должен быть инициализирован только один раз. Статические

элементы данных всех основных (фундаментальных) типов инициализируются по умолчанию значением `0`. Каждый константный статический элемент данных (`static const`) может иметь внутриклассовый инициализатор. Начиная с C++17, внутриклассовые инициализаторы можно использовать и для неконстантных элементов статических данных, добавляя в объявление ключевое слово `inline` (илл. 9.33). Если статический элемент данных является объектом класса, содержащего конструктор по умолчанию, этот элемент данных не надо инициализировать явно, потому что будет вызван его конструктор по умолчанию.

Доступ к статическим элементам данных

Статические элементы класса существуют даже при отсутствии объектов этого класса. Для доступа к открытым (`public`) статическим элементам данных и функциям класса просто указывайте перед именем нужного вам элемента имя класса и оператор разрешения области видимости (`::`). Например, если статическая переменная `martianCount` из класса `Martian` является открытой, к ней можно получить доступ по полному имени `Martian::martianCount`, даже если никаких объектов `Martian` не было создано.

Доступ к скрытым (`private`) и защищенным (`protected`, глава 10) статическим элементам класса обычно осуществляется через открытые функции класса или дружественные функции. Чтобы при отсутствии объектов класса получать доступ к его скрытым статическим (`private static`) и защищенным статическим (`protected static`) элементам данных, определите открытую статическую (`static`) функцию класса и вызывайте ее, добавляя перед ее именем имя класса и оператор разрешения области видимости. Статическая функция выполняет задачи для всего класса, а не для отдельных объектов этого класса.

Пример использования статических элементов данных

В следующем примере кода мы продемонстрируем скрытый статический элемент данных `m_count`, инициализированный значением `0` (илл. 9.33, строка 22), и открытую статическую функцию `getCount` (илл. 9.33, строка 16). Статический элемент данных также можно инициализировать в файле реализации класса. Например, мы могли бы поместить в файл `Employee.cpp` (илл. 9.34) после включения заголовка `Employee.h` такую инструкцию:

```
int Employee::count{0};
```

```
 1 // Fig. 9.33: Employee.h
 2 // Определение класса Employee со статическим элементом данных,
 3 // отслеживающим количество объектов класса Employee
 4 #pragma once // Предотвращаем повторные включения заголовка
 5 #include <string>
 6 #include <string_view>
 7
 8 class Employee {
 9 public:
10    Employee(std::string_view firstName, std::string_view lastName);
```

```
11    ~Employee(); // Деструктор
12    const std::string& getFirstName() const; // Возвращаем имя сотрудника
13    const std::string& getLastName() const; // Возвращаем фамилию сотрудника
14
15    // Статическая функция класса
16    static int getCount(); // Возвращаем количество созданных объектов
17 private:
18    std::string m_firstName;
19    std::string m_lastName;
20
21    // Статические данные
22    inline static int m_count{0}; // Количество созданных объектов
23 };
```

Илл. 9.33 | Определение класса Employee со статическим элементом данных, отслеживающим количество объектов класса Employee

На илл. 9.34 в строке 10 определяется статическая функция **getCount**. Обратите внимание, что в этом объявлении нет ключевого слова **static**, которое не может применяться к определению элемента вне определения класса. В этой программе элемент данных **m_count** является счетчиком объектов класса **Employee**, имеющихся в памяти в данный момент. Когда объекты класса **Employee** существуют, их функции могут обращаться к элементу **m_count**, как показано в конструкторе (строка 16) и в деструкторе (строка 25).

```
 1 // Fig. 9.34: Employee.cpp
 2 // Определения функций класса Employee
 3 #include <fmt/format.h>
 4 #include <iostream>
 5 #include "Employee.h" // Определение класса Employee
 6 using namespace std;
 7
 8 // Определение статической функции, возвращающей количество созданных
 9 // объектов класса Employee (она объявлена как static в Employee.h)
10 int Employee::getCount() {return m_count;}
11
12 // Конструктор инициализирует нестатические элементы данных и
13 // выполняет инкремент статического счетчика
14 Employee::Employee(string_view firstName, string_view lastName)
15    : m_firstName(firstName), m_lastName(lastName) {
16    ++m_count; // Увеличиваем статический счетчик количества сотрудников
17    std::cout << fmt::format("Employee constructor called for {} {}\n",
18       m_firstName, m_lastName);
19 }
20
21 // Деструктор выполняет декремент статического счетчика
22 Employee::~Employee() {
23    std::cout << fmt::format("~Employee() called for {} {}\n",
24       m_firstName, m_lastName);
25    --m_count; // Уменьшаем статический счетчик количества сотрудников
```

Окончание ⇨

```
26 }
27
28 // Возвращаем имя сотрудника
29 const string& Employee::getFirstName() const {return m_firstName;}
30
31 // Возвращаем фамилию сотрудника
32 const string& Employee::getLastName() const {return m_lastName;}
```

Илл. 9.34 | Определения функций класса Employee

В коде на илл. 9.35 статическая функция `getCount` используется, чтобы отслеживать количество имеющихся в памяти объектов класса `Employee` в разные моменты выполняемой программы. Программа вызывает функцию `Employee::getCount()`:

- до создания каких-либо объектов класса `Employee` (строка 11);
- после создания двух объектов класса `Employee` (строка 23);
- после уничтожения всех объектов класса `Employee` (строка 34).

Когда объекты класса `Employee` находятся в области видимости, вы можете вызывать функцию `getCount` и для них тоже. Например, в строке 23 вы могли бы использовать такой вызов:

```
e1.getCount()
```

Или такой:

```
e2.getCount()
```

Каждый из этих вызовов возвращает текущее значение статического элемента `m_count` класса `Employee`.

```
 1 // fig09_35.cpp
 2 // Отслеживание количества объектов класса в статическом элементе класса
 3 #include <fmt/format.h>
 4 #include <iostream>
 5 #include "Employee.h" // Определение класса Employee
 6
 7 int main() {
 8    // Объекты еще не созданы; вызываем статическую функцию getCount
 9    // через имя класса и оператор разрешения области видимости
10    std::cout << fmt::format("Initial employee count: {}\n",
11       Employee::getCount()); // Используем имя класса
12
13    // В следующем блоке мы будем создавать объекты класса Employee
14    // и уничтожать их до завершения функции main
15    {
16       const Employee e1{"Susan", "Baker"};
17       const Employee e2{"Robert", "Jones"};
18
19       // Есть два объекта; снова вызываем статическую функцию getCount
```

```
20       // через имя класса и оператор разрешения области видимости
21       std::cout << fmt::format(
22       "Employee count after creating objects: {}\n\n",
23       Employee::getCount());
24
25       std::cout << fmt::format("Employee 1: {} {}\nEmployee 2: {} {}\n\n",
26          e1.getFirstName(), e1.getLastName(),
27          e2.getFirstName(), e2.getLastName());
28    }
29
30    // Нет ни одного объекта; снова вызываем статическую функцию getCount
31    // через имя класса и оператор разрешения области видимости
32    std::cout << fmt::format(
33       "Employee count after objects are deleted: {}\n",
34       Employee::getCount());
35 }
```

```
Initial employee count: 0
Employee constructor called for Susan Baker
Employee constructor called for Robert Jones
Employee count after creating objects: 2

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Robert Jones
~Employee() called for Susan Baker
Employee count after objects are deleted: 0
```

Илл. 9.35 | Отслеживание количества объектов класса в статическом элементе класса

В строках 15–28 определяется вложенный блок в функции `main`. Напомним, что локальные переменные существуют до тех пор, пока существует область видимости, в которой они определены. Во вложенном блоке мы создаем два объекта класса `Employee` (строки 16–17). При создании каждого из них конструктор выполняет инкремент статического элемента данных `count` класса `Employee`. Когда программа достигает строки 28, оба объекта уничтожаются. Для этого вызываются их деструкторы, каждый из которых выполняет декремент статического элемента данных `count` класса `Employee`.

Дополнительные сведения о статических функциях

Функцию класса надо объявлять как статическую, если она не обращается к нестатическим элементам данных или к нестатическим функциям класса. Статическая функция не имеет указателя `this`, поскольку статические элементы данных и статические функции существуют независимо от любых объектов класса. Указатель `this` должен ссылаться на конкретный объект, но статическую функцию можно вызывать даже при отсутствии объектов ее класса. Использование указателя `this` в статической функции класса вызовет ошибку компиляции.

Статическую функцию класса нельзя объявлять как константную. Квалификатор `const` означает, что функция не может изменять содержимое объекта. Но статические функции существуют и работают независимо от любых объектов класса. Объявление статической функции с квалификатором `const` вызовет ошибку компиляции.

9.21. Агрегаты

Определение агрегатов сформулировано в разделе 9.4.1 стандарта C++:

http://wg21.link/n4861

Согласно этому документу, **агрегат (aggregate)** — это массив или класс, который:

- не имеет пользовательских конструкторов;
- не имеет скрытых (`private`) или защищенных (`protected`, глава 10) нестатических элементов данных;
- не имеет виртуальных функций (глава 10);
- не имеет скрытых, защищенных или виртуальных базовых классов (глава 10).

Требование об отсутствии пользовательских конструкторов было введено в C++20 для того, чтобы исключить возможность инициализации агрегатного объекта в обход пользовательского конструктора[1].

Вы можете определить агрегат как класс, в котором все данные являются открытыми (`public`). Агрегатом может быть обычный класс или структура. Структура (`struct`) — это класс, который по умолчанию содержит только открытые элементы. Следующая структура определяет агрегатный тип с именем `Record`, содержащий четыре открытых элемента данных:

```
struct Record {
   int account;
   string first;
   string last;
   double balance;
};
```

В C++ Core Guidelines рекомендуется использовать `class`, а не `struct`, если хотя бы один элемент данных или хотя бы одна функция не должны быть открытыми[2].

---

[1] «Prohibit Aggregates with User-Declared Constructors». Был доступен: 15 сентября 2023 г. http://wg21.link/p1008.

[2] C++ Core Guidelines, «C.8: Use class Rather Than struct if Any Member Is Non-Public». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-class.

9.21.1. Инициализация агрегата

Вы можете инициализировать объект агрегатного типа `Record` таким образом:

```
Record record{100, "Brian", "Blue", 123.45};
```

В C++11 нельзя было использовать скобочную инициализацию для объекта агрегатного типа, если любой из его нестатических элементов данных объявлялся с внутриклассовым инициализатором. Например, приведенная выше инструкция привела бы к ошибке компиляции, если бы тип `Record` был определен со значением по умолчанию для `balance`:

```
struct Record {
   int account;
   std::string first;
   std::string last;
   double balance{0.0};
};
```

Начиная с C++14, этого ограничения нет. Кроме того, в вашем списке инициализаторов объекта агрегатного типа может быть меньше инициализаторов, чем элементов данных в объекте, например:

```
Record record{0, "Brian", "Blue"};
```

В этом случае все остальные элементы данных будут инициализированы автоматически:

- Элементы данных с внутриклассовыми инициализаторами получат их значения (в предыдущем примере элемент `balance` в объекте `record` получит значение `0.0`).
- Элементы данных без внутриклассовых инициализаторов будут инициализированы пустыми фигурными скобками (`{}`). Это означает, что переменные основных (фундаментальных) типов получат значение `0`, переменные типа `bool` — значение `false`, а объекты — значения по умолчанию.

9.21.2. Назначенные инициализаторы (C++20)

Начиная с C++20, агрегаты поддерживают **назначенные инициализаторы (designated initializers)**, в которых вы можете указывать по именам, какие элементы данных надо инициализировать. Используя предыдущее определение структуры `Record`, вы можете инициализировать объект `record` таким образом:

```
Record record{.first{"Sue"}, .last{"Green"}};
```

В этом случае будут явно инициализированы только два элемента данных из четырех. Перед каждым явно указанным именем элемента ставится точка (`.`). Все указанные вами идентификаторы должны быть перечислены в том же порядке, в каком они объявлены в агрегатном типе. Предыдущая инструкция инициализи-

рует элементы `first` и `last` значениями `"Sue"` и `"Green"` соответственно. Остальные элементы получат значения по умолчанию:

- `account` получит значение `0`;
- `balance` получит значение из объявления типа, в данном случае — `0.0`.

Другие преимущества назначенных инициализаторов[1]

Добавление новых элементов данных к агрегатному типу не приведет к нарушению существующих инструкций, использующих назначенные инициализаторы. Все новые элементы, которые не инициализированы явно, просто получат значения по умолчанию. Кроме того, назначенные инициализаторы улучшают совместимость с языком C, в котором они используются, начиная с версии C99.

9.22. Готовые объекты: сериализация в формате JSON

Современные программы все чаще работают «в облаке», то есть их фрагменты распределяются по интернету. Многие приложения, которыми вы ежедневно пользуетесь, взаимодействуют через интернет с **облачными сервисами (cloud-based services)**, для работы которых обычно требуются огромные кластеры компьютерных ресурсов (таких, как процессоры, память, жесткие диски и базы данных).

Сервис, предоставляющий пользователям доступ через интернет, называется **веб-сервисом (web service)**. Многие пользовательские приложения взаимодействуют с веб-сервисами, обмениваясь объектами в формате JSON. **JSON (JavaScript Object Notation)** — это основанный на тексте формат обмена данными, представляющий каждый передаваемый объект в виде набора пар *имя — значение*. Текст в этом формате может прочитать как компьютер, так и человек. В последние годы JSON стал наиболее предпочтительным форматом данных для передачи объектов на разных платформах.

Формат данных JSON

Каждый объект JSON содержит список пар *имя — значение*, разделенный запятыми и заключенный в фигурные скобки. Например, из трех таких пар может состоять запись о клиенте в базе данных банка:

```
{"account": 100, "name": "Jones", "balance": 24.98}
```

JSON также поддерживает массивы. Массив записывается в виде заключенного в квадратные скобки списка значений, разделенных запятыми. Например, так выглядит массив из трех целых чисел в формате JSON:

```
[100, 200, 300]
```

[1] «Designated Initialization». Был доступен: 15 сентября 2023 г. http://wg21.link/p0329r0.

JSON может хранить в объектах и массивах:

- строки в кавычках (например, `"Jones"`);
- числа (например, `100` или `24.98`);
- логические значения JSON (представленные как `true` или `false`);
- `null` (отсутствие значения);
- массивы, содержащие любые допустимые значения JSON;
- другие объекты JSON.

Массив JSON может содержать элементы одного или разных типов.

Сериализация

Преобразование объекта в другой формат для сохранения в виде файла или для передачи через интернет называется **сериализацией (serialization)**. Восстановление объекта из сериализованных данных называется **десериализацией (deserialization)**. JSON — это один из нескольких общепринятых форматов сериализации. Другим распространенным форматом является XML (eXtensible Markup Language).

Безопасность сериализации

В некоторых языках программирования встроены собственные механизмы и форматы сериализации. Десериализация объектов, преобразованных в такие форматы, может создавать опасные проблемы. По данным Open Web Application Security Project (OWASP), собственные механизмы сериализации «могут быть использованы для вредоносных воздействий при обработке непроверенных данных. Были обнаружены воздействия на десериализаторы, позволяющие осуществлять атаки типа "отказ в обслуживании", "контроль доступа" и "удаленное выполнение кода"»[1]. OWASP также сообщает, что можно значительно снизить риск атак, отказавшись от имеющихся в языках программирования собственных форматов сериализации в пользу форматов «чистых данных», таких как JSON или XML.

Заголовочная библиотека cereal

Заголовочная библиотека `cereal`[2] предназначена для сериализации и десериализации объектов в формате JSON (который мы продемонстрируем), а также в формате XML и в двоичных форматах. Эта библиотека поддерживает основные (фундаментальные) типы данных и может обрабатывать большинство типов стандартной библиотеки C++, если вы включите в программу соответствующие заголовки `cereal` для каждого типа. Как вы увидите в следующем разделе, библио-

---

[1] «Deserialization Cheat Sheet». Был доступен: 15 сентября 2023 г. https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html.

[2] © 2017. У. Шейн Грант (W. Shane Grant) и Рэндольф Вурхис (Randolph Voorhies), «cereal — A C++11 library for serialization». http://uscilab.github.io/cereal.

тека cereal поддерживает и пользовательские типы. Документация к библиотеке cereal опубликована здесь:

https://uscilab.github.io/cereal/index.html

Для вашего удобства мы добавили библиотеку cereal к примерам кода из этой книги в папку libraries. Вы должны указать в параметрах IDE или компилятора папку include этой библиотеки, как и в предыдущих примерах с готовыми объектами.

9.22.1. Сериализация вектора объектов, содержащего открытые данные

Начнем с сериализации объектов, содержащих только открытые данные. В следующей программе, разделенной на части для удобства обсуждения (илл. 9.36), мы:

- создаем вектор (vector) объектов типа Record и отображаем его содержимое;
- используем библиотеку cereal для сериализации этого вектора в виде текстового файла;
- десериализуем содержимое этого файла обратно в вектор объектов Record и отображаем его содержимое.

Для сериализации в формате JSON нужен заголовок json.hpp (строка 3). Для сериализации объекта типа std::vector нужен заголовок vector.hpp (строка 4).

```
1 // fig09_36.cpp
2 // Сериализация и десериализация объектов с помощью библиотеки cereal
3 #include <cereal/archives/json.hpp>
4 #include <cereal/types/vector.hpp>
5 #include <fmt/format.h>
6 #include <fstream>
7 #include <iostream>
8 #include <vector>
9
```

Илл. 9.36 | Сериализация и десериализация объектов с помощью библиотеки cereal

Агрегатный тип Record

Record — это агрегатный тип данных, определенный как структура (struct). Напомним, что все данные в агрегате должны быть открытыми. В структуре это требование выполняется по умолчанию.

```
10 struct Record {
11    int account{};
12    std::string first{};
13    std::string last{};
14    double balance{};
15 };
16
```

Функция serialize для работы с объектами Record

Библиотека `cereal` предоставляет несколько способов сериализации объектов. Если тип объектов, которые вы хотите сериализовать, содержит только открытые (`public`) данные, вам достаточно определить шаблон функции `serialize` (строки 19–25) с двумя параметрами. Первый параметр — архив (`Archive`), куда будет помещен результат сериализации, а второй параметр — объект вашего типа (`Record`)[1]. Мы будем вызывать функцию `serialize` и для сериализации, и для десериализации объектов `Record`. Шаблон функции позволяет вам выбирать между форматами сериализации и десериализации (JSON, XML или двоичные форматы), передавая объект соответствующего типа архива `cereal`. Библиотека предоставляет реализации архива для каждого случая.

```
17 // Шаблон функции serialize для сериализации объектов Record в объект Archive
18 // и для десериализации из объекта Archive в объекты Record
19 template <typename Archive>
20 void serialize(Archive& archive, Record& record) {
21    archive(cereal::make_nvp("account", record.account),
22       cereal::make_nvp("first", record.first),
23       cereal::make_nvp("last", record.last),
24       cereal::make_nvp("balance", record.balance));
25 }
26
```

Каждый тип архива `cereal` имеет перегруженный оператор «круглые скобки», позволяющий использовать имя архива в качестве имени функции (строки 21–24). В зависимости от выполняемого действия — сериализации или десериализации объектов `Record` — эта функция будет:

- либо выводить содержимое объектов `Record` в указанный поток;
- либо записывать сериализованные данные из указанного потока, создавая объекты `Record`.

Каждый вызов `cereal::make_nvp`[2], например в строке 21:

```
cereal::make_nvp("account", record.account)
```

предназначен для сериализации. Он создает пару *имя — значение*, в которой первый аргумент (в данном случае `"account"`) будет именем, а второй аргумент (в данном случае `record.account` типа `int`) будет значением. Давать осмысленные имена значениям не обязательно, но такие имена делают выходные данные в формате JSON более понятными для человека, как вы скоро увидите. Если вы не дали значениям имена, библиотека `cereal` использует стандартные имена, например `value0`, `value1` и т. д.

---

[1] Функцию serialize также можно определить как шаблон функции класса с единственным параметром Archive: https://uscilab.github.io/cereal/serialization_functions.html.

[2] Имя make_nvp — это сокращение от «make name-value pair» («создать пару имя — значение»).

Функция displayRecords

Мы включили в программу функцию `displayRecords`, чтобы показать содержимое объектов `Record` до сериализации и после десериализации. Функция выводит на экран элементы вектора, который она получает в качестве аргумента:

```
27 // Отображаем объекты вектора
28 void displayRecords(const std::vector<Record>& records) {
29     for (const auto& r : records) {
30         std::cout << fmt::format("{} {} {} {:.2f}\n",
31             r.account, r.first, r.last, r.balance);
32     }
33 }
34
```

Создание объектов Record для их последующей сериализации

В функции `main` в строках 36–39 мы создаем вектор с именем `records` и инициализируем его двумя объектами агрегатного типа `Record` (строки 37 и 38). С помощью CTAD компилятор автоматически определяет тип элементов вектора по типам из списков инициализации объектов `Record`. В строке 42 мы выводим на экран содержимое вектора, чтобы убедиться, что объекты `Record` инициализированы правильно.

```
35 int main() {
36     std::vector records{
37         Record{100, "Brian", "Blue", 123.45},
38         Record{200, "Sue", "Green", 987.65}
39     };
40
41     std::cout << "Records to serialize:\n";
42     displayRecords(records);
43
```

```
Records to serialize:
100 Brian Blue 123.45
200 Sue Green 987.65
```

Сериализация объектов Record с помощью класса cereal::JSONOutputArchive

Класс `cereal::JSONOutputArchive` сериализует данные в формате JSON в указанный поток, например в стандартный поток вывода или в файловый поток. В строке 45 мы пробуем открыть файл `records.json` для записи. Если файл удалось открыть, в строке 46 создается объект `cereal::JSONOutputArchive` с именем `archive`. Мы инициализируем его объектом `ofstream` с именем `output`, чтобы записать в файл преобразованные в формат JSON данные. В строке 47 объект `archive` используется для вывода пары *имя — значение*, где имя — `"records"`, а значение — наш вектор `records`. В процессе сериализации вектора сериализуются

все его элементы, то есть выполняется вызов функции `serialize` (строки 19–25) для каждого объекта `Record` в векторе.

```
44    // Сериализуем вектор объектов Records в формате JSON и сохраняем в файле
45    if (std::ofstream output{"records.json"}) {
46       cereal::JSONOutputArchive archive{output};
47       archive(cereal::make_nvp("records", records)); // Сериализуем records
48    }
49
```

Содержимое файла records.json

После выполнения инструкции в строке 47 файл `records.json` содержит следующие данные в формате JSON:

```
{
    "records": [
        {
            "account": 100,
            "first": "Brian",
            "last": "Blue",
            "balance": 123.45
        },
        {
            "account": 200,
            "first": "Sue",
            "last": "Green",
            "balance": 987.65
        }
    ]
}
```

Внешние фигурные скобки обозначают весь документ JSON. На иллюстрации мы выделили темно-серым прямоугольником пару *имя — значение* с именем `"records"`, значением которой является массив JSON. Он состоит из двух объектов JSON, которые мы выделили внутренними прямоугольниками (каждый объект JSON заключается в фигурные скобки). Таким образом, в этом массиве хранятся преобразованные в формат JSON элементы вектора `records`, который мы сериализовали в строке 47. Каждый объект JSON в массиве содержит четыре пары *имя — значение*, сериализованные функцией `serialize` из соответствующего объекта `Record`.

Десериализация объектов Record с помощью класса cereal::JSONInputArchive

Теперь мы десериализуем данные из файла и запишем их в новый вектор объектов `Record`. Чтобы библиотека `cereal` могла воссоздавать объекты в памяти, ей нужен доступ к конструктору по умолчанию для каждого типа. Библиотека будет вызывать конструктор для создания объекта, а затем заполнять нужными

значениями элементы данных этого объекта. Для структур, как и для классов, компилятор тоже предоставляет открытый (`public`) конструктор по умолчанию, если пользовательский конструктор не определен.

```
50    // Десериализуем данные JSON из файла в вектор объектов Records
51    if (std::ifstream input{"records.json"}) {
52       cereal::JSONInputArchive archive{input};
53       std::vector<Record> deserializedRecords{};
54       archive(deserializedRecords); // Десериализуем records
55       std::cout << "\nDeserialized records:\n";
56       displayRecords(deserializedRecords);
57    }
58 }
```

```
Deserialized records:
100 Brian Blue 123.45
200 Sue Green 987.65
```

Класс `cereal::JSONInputArchive` десериализует данные в формате JSON из указанного потока. В строке 51 мы пробуем открыть файл `records.json` для чтения. Если файл удалось открыть, в строке 52 создается объект `cereal::JSONInputArchive` с именем `archive`. Мы инициализируем его объектом `ifstream` с именем `input`, чтобы `archive` мог прочитать из файла `records.json` данные в формате JSON. В строке 53 мы создаем пустой вектор объектов `Record`, в который будем записывать данные из файла. В строке 54 объект `archive` используется для десериализации данных из файла в объект `deserializedRecords`. В процессе десериализации вектора десериализуются все его элементы. Для этого снова вызывается функция `serialize` (строки 19–25), но поскольку `archive` является объектом `cereal::JSONInputArchive`, каждый вызов функции `serialize` считывает данные из одного объекта JSON, создает объект `Record` и записывает в него эти данные.

9.22.2. Сериализация вектора объектов, содержащего скрытые данные

Объекты со скрытыми (`private`) данными тоже можно сериализовать. Для этого надо объявить функцию `serialize` как дружественную классу, чтобы она получила доступ к его скрытым данным. Чтобы продемонстрировать сериализацию скрытых данных, мы взяли копию предыдущей программы (илл. 9.36) и внесли в нее соответствующие изменения (илл. 9.37), в том числе заменили определение агрегата `Record` определением класса `Record` в строках 12–35.

```
1 // fig09_37.cpp
2 // Сериализация и десериализация объектов, содержащих скрытые данные
3 #include <cereal/archives/json.hpp>
4 #include <cereal/types/vector.hpp>
5 #include <fmt/format.h>
6 #include <fstream>
7 #include <iostream>
```

```
#include <string>
#include <string_view>
#include <vector>

class Record {
   // Объявляем функцию serialize дружественной для доступа к скрытым данным
   template<typename Archive>
   friend void serialize(Archive& archive, Record& record);

public:
   // Конструктор
   explicit Record(int account = 0, std::string_view first = "",
      std::string_view last = "", double balance = 0.0)
      : m_account{account}, m_first{first},
        m_last{last}, m_balance{balance} {}

   // Get-функции класса
   int getAccount() const {return m_account;}
   const std::string& getFirst() const {return m_first;}
   const std::string& getLast() const {return m_last;}
   double getBalance() const {return m_balance;}

private:
   int m_account{};
   std::string m_first{};
   std::string m_last{};
   double m_balance{};
};

// Шаблон функции serialize для сериализации объектов Record в объект Archive
// и для десериализации из объекта Archive в объекты Record
template <typename Archive>
void serialize(Archive& archive, Record& record) {
   archive(cereal::make_nvp("account", record.m_account),
      cereal::make_nvp("first", record.m_first),
      cereal::make_nvp("last", record.m_last),
      cereal::make_nvp("balance", record.m_balance));
}

// Отображаем объекты вектора
void displayRecords(const std::vector<Record>& records) {
   for (const auto& r : records) {
      std::cout << fmt::format("{} {} {} {:.2f}\n", r.getAccount(),
         r.getFirst(), r.getLast(), r.getBalance());
   }
}

int main() {
   std::vector records{
      Record{100, "Brian", "Blue", 123.45},
      Record{200, "Sue", "Green", 987.65}
   };
```

Окончание ⇨

```
60
61    std::cout << "Records to serialize:\n";
62    displayRecords(records);
63
64    // Сериализуем вектор объектов Records в формате JSON и сохраняем в файле
65    if (std::ofstream output{"records2.json"}) {
66       cereal::JSONOutputArchive archive{output};
67       archive(cereal::make_nvp("records", records)); // Сериализуем records
68    }
69
70    // Десериализуем данные JSON из файла в вектор объектов Records
71    if (std::ifstream input{"records2.json"}) {
72       cereal::JSONInputArchive archive{input};
73       std::vector<Record> deserializedRecords{};
74       archive(deserializedRecords); // Десериализуем records
75       std::cout << "\nDeserialized records:\n";
76       displayRecords(deserializedRecords);
77    }
78 }
```

Илл. 9.37 | Сериализация и десериализация объектов, содержащих скрытые данные

Класс `Record` содержит конструктор (строки 19–22), *get*-функции для получения данных (строки 25–28) и скрытые данные (строки 31–34). Отметим два ключевых момента:

- В строках 14–15 объявляется шаблон функции `serialize`, дружественной для этого класса. Дружественная функция `serialize` получает прямой доступ к скрытым элементам данных класса: `account`, `first`, `last` и `balance`.
- Все параметры конструктора имеют аргументы по умолчанию, что позволяет библиотеке `cereal` использовать его в качестве конструктора по умолчанию при десериализации объектов `Record`.

Функция `serialize` (строки 39–45) обращается к скрытым элементам данных класса `Record`, а функция `displayRecords` (строки 48–53) использует *get*-функции класса для доступа к скрытым данным и для их отображения на экране. Функция `main` здесь такая же, как в разделе 9.22.1. Она дает такие же результаты, поэтому мы не приводим выходные данные.

9.23. Итоги

В этой главе вы создавали пользовательские классы и объекты этих классов, а затем вызывали их функции для выполнения нужных действий. Вы объявляли элементы данных класса, предназначенные для хранения данных в объектах, и определяли функции класса, предназначенные для обработки этих данных. Вы узнали, как использовать конструктор класса, чтобы присваивать начальные значения элементам данных ваших объектов.

На примере учебного класса `Time` мы показали целый ряд дополнительных возможностей. Вы спроектировали класс, отделив его интерфейс от его реализации. Вы обращались к элементам данных объекта через указатель на объект с помощью оператора «стрелка». Вы узнали, что функции класса имеют область видимости класса: имя любой функции класса известно только другим функциям этого класса, за исключением тех случаев, когда клиент класса вызывает ее через имя объекта, ссылку на объект класса, указатель на объект класса или оператор разрешения области видимости. Мы рассмотрели функции доступа (обычно применяемые для получения данных и для проверки истинности условий) и служебные функции (скрытые функции класса, которые поддерживают работу открытых функций).

Вы узнали, что конструктор может содержать аргументы по умолчанию, что позволяет вызывать его несколькими способами. Вы также узнали, что любой конструктор, который может быть вызван без аргументов, является конструктором по умолчанию. Мы показали, как распределять код между конструкторами с помощью делегирующих конструкторов. Мы рассмотрели деструкторы, выполняющие завершающую обработку перед уничтожением объектов, и продемонстрировали порядок вызова конструкторов и деструкторов объекта.

На примере неправильного кода мы показали проблемы, которые возникают, когда функция класса возвращает ссылку или указатель на скрытый элемент данных, нарушая инкапсуляцию класса. Мы также показали, что объекты одного типа можно присваивать друг другу с помощью оператора присваивания по умолчанию.

Вы узнали, как определять константные объекты и функции класса для защиты от некорректного изменения объектов, обеспечивая тем самым соблюдение принципа наименьших привилегий. Вы познакомились с понятием композиции, благодаря которой элементами класса могут быть объекты других классов. Вы научились объявлять и использовать дружественные функции.

Теперь вы знаете, что указатель `this` передается в качестве неявного аргумента всем нестатическим функциям класса, позволяя им обращаться к элементам данных нужного объекта и к другим нестатическим функциям класса. Мы явно использовали указатель `this` для доступа к элементам класса и для каскадного вызова функций класса. Мы объяснили необходимость статических элементов данных и функций класса, а затем показали, как их объявлять и использовать.

Вы познакомились с понятием агрегатных типов данных и с назначенными инициализаторами (C++20) для них. В очередном примере работы с готовыми объектами вы выполнили сериализацию объектов в формате JSON (JavaScript Object Notation) с помощью библиотеки `cereal`.

В следующей главе мы продолжим обсуждение классов и познакомим вас с наследованием. Вы увидите, что классы могут наследовать свои атрибуты и поведение от общего для них класса, который называется базовым. Затем мы расширим тему наследования и введем понятие полиморфизма. Эта объектно-ориентированная концепция позволяет писать программы, в которых единообразно обрабатываются объекты разных классов, связанных наследованием.

10

ООП: наследование и динамический полиморфизм

В этой главе вы:

- узнаете традиционные и современные идиомы[1] наследования, познакомитесь с базовыми и производными классами;
- познакомитесь с порядком вызова конструкторов и деструкторов в иерархиях наследования;
- узнаете, как динамический полиморфизм упрощает программирование и способствует расширяемости систем;
- познакомитесь с ключевым словом `override`, предназначенным для переопределения виртуальных функций базового класса;
- научитесь запрещать спецификатором `final` переопределение функций;
- научитесь запрещать спецификатором `final` использование класса в качестве базового;
- научитесь наследованию с абстрактными и конкретными классами;
- узнаете, как в C++ реализованы ресурсоемкие виртуальные функции и динамическое связывание;
- используете идиому невиртуального интерфейса (NVI) с открытыми невиртуальными функциями и со скрытыми или защищенными виртуальными функциями;
- используете интерфейсы для создания более гибких систем на основе динамического полиморфизма;
- реализуете метод динамического полиморфизма без иерархии классов с помощью `std::variant` и `std::visit`.

[1] Идиома языка программирования — устойчивая синтаксическая конструкция, объединенная общей целью. — *Примеч. ред.*

10.1. Введение

В этой главе продолжается изучение объектно-ориентированного программирования (ООП), вводятся понятия наследования и динамического полиморфизма. С помощью **наследования (inheritance)** вы будете создавать классы, которые обладают функциональностью готовых (ранее созданных) классов, но дают вам возможность расширять и адаптировать эту функциональность под свои задачи.

Создавая класс, вы можете указать, что он должен **наследовать (inherit)** элементы другого класса. Этот существующий класс называется **базовым классом (base class)**, а ваш новый класс называется **производным классом (derived class)**. В некоторых языках программирования, таких как Java и C#, есть термины **суперкласс (superclass)** и **подкласс (subclass)**, обозначающие базовый и производный классы соответственно[1].

Отношения «содержит» и «является»

Вы должны различать отношения «содержит» (имеет) и «является» (представляет собой):

- Отношение **«содержит» («has-a»)** — это композиция (раздел 9.17). Объект содержит в качестве своих элементов один или несколько объектов других классов. Например, автомобиль содержит руль, содержит педаль тормоза, содержит двигатель, содержит коробку передач и т. д.
- Отношение **«является» («is-a»)** — это наследование. Объект производного класса может рассматриваться как объект, имеющий тип базового класса. Например, автомобиль является транспортным средством, то есть демонстрирует поведение и атрибуты транспортного средства.

Динамический полиморфизм

Мы объясним и покажем на примерах, что такое динамический полиморфизм с иерархиями наследования[2]. **Динамический полиморфизм (runtime polymorphism)** позволяет удобно «программировать вообще», а не «программировать в частности». Программы могут обрабатывать объекты разных классов, связанных наследованием, как если бы все эти объекты имели один тип базового класса. Вы увидите, что полиморфный код (который мы сначала реализуем с помощью наследования и виртуальных функций) обращается к объектам через указатели базового класса или ссылки базового класса.

---

[1] Иногда программисты называют базовые классы родительскими, а производные – дочерними. — *Примеч. ред.*

[2] Позже вы увидите, что полиморфизм не обязательно реализуется с помощью иерархий наследования. Мы рассмотрим статический полиморфизм в главе 15.

Простое расширение программ

С помощью динамического полиморфизма вы можете проектировать и внедрять большие программы, функциональность которых легко расширяется по мере необходимости. Вы можете добавлять новые классы, почти не внося изменений в основные части вашей программы, при условии, что эти новые классы входят в иерархию наследования вашей программы. Вы можете модифицировать только тот код, в котором потребуется прямое взаимодействие с новыми классами. Предположим, что от класса `Vehicle` наследуется новый класс `Car`. Вам нужно написать только этот новый класс и код, который создает объекты класса `Car` и добавляет их в систему. После этого новый класс просто «подключится» к программе, а весь основной код, обрабатывающий объекты класса `Vehicle`, может остаться прежним.

Динамический полиморфизм и виртуальные функции «под капотом» компилятора

В этой главе мы рассмотрим, как устроены в языке C++ динамический полиморфизм, виртуальные функции и динамическое связывание. Стандарт C++ не определяет, каким образом они должны быть реализованы в компиляторе. Мы подробно расскажем об одном из возможных вариантов их реализации.

Мы начнем с традиционного введения в наследование и динамический полиморфизм, чтобы вы познакомились с основными идеями на базовом и среднем уровне и понимали, как работают эти технологии. Далее в этой главе (и в последующих главах) мы рассмотрим современные идиомы программирования.

Идиома невиртуального интерфейса

В частности, мы покажем вам другой подход к динамическому полиморфизму: **идиому невиртуального интерфейса (non-virtual interface idiom**, или сокращенно **NVI**), требующую, чтобы каждая функция базового класса предназначалась только для одной из двух целей:

- или эта функция вызывается клиентским кодом для выполнения задачи;
- или это функция, которую могут настраивать (адаптировать для своих целей) производные классы.

Настраиваемые функции — это внутренние детали реализации классов, облегчающие обслуживание и расширение цифровых систем без необходимости внесения изменений в код клиентских приложений.

Интерфейсы и внедрение зависимостей

Чтобы объяснить механику наследования, мы сосредоточили первые примеры этой главы на **наследовании реализации (implementation inheritance)**, которое в основном используется для определения тесно связанных классов со многими одинаково реализованными элементами данных и функциями. Этот стиль наследования десятилетиями широко практиковался в объектно-ориентированном

программировании. Но со временем выявились его слабые стороны. Язык C++ используется для создания критически важных для бизнеса цифровых систем, в том числе очень больших. Эмпирические результаты многолетнего опыта создания систем, основанных на наследовании реализации, показывают, что их сложно модифицировать.

По этой причине мы затем проведем **рефакторинг (refactoring)**, то есть доработку одного из наших примеров: заменим наследование реализации на **наследование интерфейса (interface inheritance)**. Базовый класс не будет предоставлять никаких деталей реализации. Вместо этого он будет содержать «заполнители», которые сообщают производным классам, какие функции они должны реализовать. А производные классы будут предоставлять детали реализации: элементы данных и реализованные функции классов. В рамках этого примера мы представим **внедрение зависимостей (dependency injection)**: добавим в класс указатель на объект, обеспечивающий нужное поведение объектов класса (в нашем примере — вычисление зарплаты сотрудника). Этот указатель может быть переопределен. Например, если сотрудник получает повышение, мы можем изменить расчет зарплаты, просто перенаправив указатель на соответствующий объект. Вы поймете, почему этот доработанный код более удобен для модификации.

Динамический полиморфизм с помощью std::variant и std::visit

Затем мы реализуем динамический полиморфизм для объектов классов, которые не связаны наследованием и не имеют виртуальных функций. Для этого мы будем использовать шаблон класса `std::variant` (C++17) и функцию `std::visit` стандартной библиотеки. Вызов общей функциональности объектов, типы которых не связаны между собой, называется **утиной типизацией (duck typing)**.

Множественное наследование

Мы продемонстрируем **множественное наследование (multiple inheritance)**, которое позволяет классу наследовать элементы нескольких базовых классов. Мы рассмотрим потенциальные проблемы множественного наследования и обсудим их решения с помощью виртуального наследования.

Другие способы реализации полиморфизма

Наконец, мы рассмотрим другие методы динамического полиморфизма и несколько методов статического полиморфизма на основе шаблонов. Статический полиморфизм подробнее рассматривается в главе 15. Там же вы узнаете о введенных в C++20 **концептах (concepts)**, которые делают ненужными некоторые устаревшие методы программирования и расширяют инструментарий для решения ряда задач на этапе компиляции с помощью шаблонов. Мы также дадим ссылки на дополнительные ресурсы, которые помогут вам, если вы захотите глубже изучить эти темы.

Цель главы

Цель этой главы — познакомить вас с основными принципами наследования и динамического полиморфизма, чтобы вы оценили пользу современных идиом и методов полиморфизма, которые упрощают модификацию больших программ и повышают их быстродействие.

10.2. Базовые и производные классы

В следующей таблице приведены примеры базовых и производных классов. Базовые классы, как правило, являются более **общими (general)**, а производные классы — более **конкретными (specific)**.

Базовый класс	Производные классы
`Student`	`GraduateStudent`, `UndergraduateStudent`
`Shape`	`Circle`, `Triangle`, `Rectangle`, `Sphere`, `Cube`
`Loan`	`CarLoan`, `HomeImprovementLoan`, `MortgageLoan`
`Vehicle`	`Car`, `Motorcycle`, `Boat`
`Employee`	`Faculty`, `Staff`
`Account`	`CheckingAccount`, `SavingsAccount`

Каждый объект производного класса является также и объектом типа его базового класса. Один базовый класс может иметь много производных классов. Таким образом, то множество объектов, которое представляет базовый класс, почти всегда больше множества объектов, которые представляет производный класс. Например, базовый класс `Vehicle` представляет все транспортные средства, включая легковые и грузовые автомобили, лодки, самолеты, велосипеды и т. д. Для сравнения, производный класс `Car` представляет только легковые автомобили, то есть небольшое подмножество `Vehicle`.

10.2.1. Иерархия класса CommunityMember

Наследование естественным образом формирует иерархии классов. Базовый класс существует в иерархических отношениях с производными классами. Когда классы входят в иерархии наследования, они становятся связанными с другими классами[1]. Класс может быть:

- базовым классом, предоставляющим элементы другим классам;
- производным классом, наследующим элементы от других классов;
- базовым и производным одновременно.

[1] Мы покажем, что тесно связанные классы могут затруднять модификацию систем. Мы рассмотрим альтернативы, позволяющие избежать тесной связи.

На следующей UML-диаграмме показан простой пример иерархии наследования. Такие диаграммы иллюстрируют отношения между классами в иерархии:



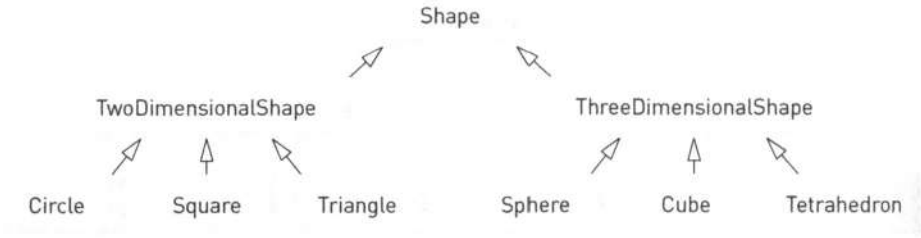
Иерархия университетского сообщества может состоять из нескольких тысяч объектов класса `CommunityMember`, таких как `Employee`, `Student` и `Alumnus`. В этой иерархии `Employee` — это `Faculty` или `Staff`, а `Faculty` — это `Administrator` или `Teacher`. Некоторые администраторы одновременно являются и преподавателями, то есть относятся к классу `AdministratorTeacher`. При одиночном (простом) наследовании производный класс наследуется от одного базового класса. При множественном наследовании производный класс наследуется от двух или более базовых классов. В этой иерархии есть пример множественного наследования: класс `AdministratorTeacher`.

Каждая стрелка на диаграмме обозначает связь **«является» («is-a»)**. На основании этих стрелок мы можем утверждать, что «`Employee` является `CommunityMember`», а «`Teacher` является `Faculty`». `CommunityMember` — это **прямой базовый класс (direct base class)** для `Employee`, `Student` и `Alumnus`. `CommunityMember` — это косвенный **базовый класс (indirect base class)** для всех остальных классов иерархии. Косвенный базовый класс находится в иерархии как минимум на два уровня выше его производных классов.

Вы можете пройти по стрелкам вверх на несколько уровней, применяя отношение «является». Таким образом, `AdministratorTeacher` является `Administrator` (а также является `Teacher`), является `Faculty`, является `Employee` и является `CommunityMember`.

10.2.2. Иерархия класса Shape и открытое наследование

Давайте рассмотрим иерархию наследования класса `Shape`, представляющего геометрические фигуры. Эта иерархия начинается с базового класса `Shape`:



Классы `TwoDimensionalShape` и `ThreeDimensionalShape` происходят от `Shape`, поэтому `TwoDimensionalShape` является `Shape` и `ThreeDimensionalShape` тоже является `Shape`. Третий уровень иерархии конкретизирует классы `TwoDimensionalShape` и `ThreeDimensionalShape`. Вы можете проследить за стрелками, чтобы определить прямые и косвенные отношения «является». Например, `Triangle` является `TwoDimensionalShape` и `Shape`, а `Sphere` является `ThreeDimensionalShape` и `Shape`.

Следующее объявление класса указывает, что `TwoDimensionalShape` наследуется от `Shape`:

```
class TwoDimensionalShape : public Shape
```

Это **открытое наследование (public inheritance)**, которое мы будем чаще всего использовать в этой главе. При открытом наследовании открытые (`public`) элементы базового класса становятся открытыми (`public`) элементами производного класса, а защищенные (`protected`) элементы базового класса становятся защищенными (`protected`) элементами производного класса. Защищенные элементы класса мы обсудим позже. Хотя из производных классов нет прямого доступа к скрытым (`private`) элементам базового класса, эти элементы тоже наследуются и являются частями объектов производных классов. Производный класс может оперировать скрытыми элементами базового класса с помощью унаследованных открытых и защищенных функций базового класса, если эти функции предоставляют такую возможность. Мы также рассмотрим скрытое (`private`) и защищенное (`protected`) наследование (раздел 10.16).

Наследование подходит не для всех отношений между классами. Иногда более уместно отношение **«содержит» («has-a»)**. Например, если у нас есть классы `Employee`, `BirthDate` и `PhoneNumber`, было бы ошибкой сказать, что «`Employee` является `BirthDate`» или «`Employee` является `PhoneNumber`». В этом случае `Employee` содержит `BirthDate` и `PhoneNumber`.

Объекты базового класса и объекты производного класса можно рассматривать как аналогичные — их общие свойства выражены в элементах базового класса. Позже в этой главе мы рассмотрим примеры, использующие преимущества этой взаимосвязи.

10.3. Взаимосвязь между базовым и производным классами

В этом разделе мы продемонстрируем взаимосвязь между базовым и производным классами, используя иерархию наследования классов, представляющих две категории сотрудников в программе расчета зарплаты:

- сотрудникам базового класса выплачивается фиксированная еженедельная зарплата (оклад);
- сотрудникам производного класса, работающим в отделе продаж, выплачивается фиксированная еженедельная зарплата (оклад), а также комиссионное вознаграждение (в процентах от их продаж).

10.3.1. Объявление и использование класса SalariedEmployee

Давайте рассмотрим определение класса `SalariedEmployee` (илл. 10.1–10.2). В его заголовке (илл. 10.1) объявлены открытые функции класса, также называемые открытыми **сервисами (services)**:

- конструктор (строка 9);
- функция `earnings` (строка 17);
- функция `toString` (строка 18);
- открытые *get*-функции и *set*-функции, оперирующие элементами данных `m_name` и `m_salary` (объявленными в строках 20–21).

Функция `setSalary` (илл. 10.2, строки 22–28) проверяет свой аргумент перед изменением элемента данных `m_salary`.

```
1 // Fig. 10.1: SalariedEmployee.h
2 // Определение класса SalariedEmployee
3 #pragma once // Предотвращаем повторные включения заголовка
4 #include <string>
5 #include <string_view>
6
7 class SalariedEmployee {
8 public:
9    SalariedEmployee(std::string_view name, double salary);
10
11    void setName(std::string_view name);
12    std::string getName() const;
13
14    void setSalary(double salary);
15    double getSalary() const;
16
```

Окончание ⇨

```
17    double earnings() const;
18    std::string toString() const;
19 private:
20    std::string m_name{};
21    double m_salary{0.0};
22 };
```

Илл. 10.1 | Определение класса SalariedEmployee

```
 1 // Fig. 10.2: SalariedEmployee.cpp
 2 // Определения функций класса SalariedEmployee
 3 #include <fmt/format.h>
 4 #include <stdexcept>
 5 #include "SalariedEmployee.h" // Определение класса SalariedEmployee
 6
 7 // Конструктор
 8 SalariedEmployee::SalariedEmployee(std::string_view name, double salary)
 9    : m_name{name} {
10    setSalary(salary);
11 }
12
13 // Записываем имя сотрудника
14 void SalariedEmployee::setName(std::string_view name) {
15    m_name = name; // Нужна проверка
16 }
17
18 // Возвращаем имя сотрудника
19 std::string SalariedEmployee::getName() const {return m_name;}
20
21 // Записываем оклад
22 void SalariedEmployee::setSalary(double salary) {
23    if (salary < 0.0) {
24       throw std::invalid_argument("Salary must be >= 0.0");
25 }
26
27    m_salary = salary;
28 }
29
30 // Возвращаем оклад
31 double SalariedEmployee::getSalary() const {return m_salary;}
32
33 // Рассчитываем зарплату
34 double SalariedEmployee::earnings() const {return getSalary();}
35
36 // Возвращаем строковое представление объекта SalariedEmployee
37 std::string SalariedEmployee::toString() const {
38    return fmt::format("name: {}\nsalary: ${:.2f}\n", getName(),
39       getSalary());
40 }
```

Илл. 10.2 | Определения функций класса SalariedEmployee

Конструктор класса SalariedEmployee

Конструктор класса (илл. 10.2, строки 8–11) использует список инициализаторов для инициализации элемента данных `m_name`, предназначенного для имени сотрудника. Мы могли бы проверить это имя (например, убедиться, что оно имеет приемлемую длину). Конструктор вызывает функцию `setSalary` для проверки и инициализации элемента данных `m_salary`.

Функции earnings и toString класса SalariedEmployee

Функция `earnings` (строка 34) вызывает функцию `getSalary` и возвращает результат. Функция `toString` (строки 37–40) возвращает строку, содержащую информацию об объекте `SalariedEmployee`.

Тест-драйв класса SalariedEmployee

На илл. 10.3 показана программа для тест-драйва класса `SalariedEmployee`. В строке 9 создается объект класса `SalariedEmployee` с именем `employee`. В строках 12–14 демонстрируются *get*-функции созданного объекта. В строке 16 функция `setSalary` используется для изменения значения `m_salary` объекта `employee`. Затем в строках 17–18 вызывается функция `toString` для получения и вывода обновленных данных из объекта `employee`. Наконец, в строке 21 отображается обновленное значение зарплаты сотрудника.

```
1 // fig10_03.cpp
2 // Тест-драйв класса SalariedEmployee
3 #include <fmt/format.h>
4 #include <iostream>
5 #include "SalariedEmployee.h" // Определение класса SalariedEmployee
6
7 int main() {
8     // Создаем объект класса SalariedEmployee
9     SalariedEmployee employee{"Sue Jones", 300.0};
10
11    // Получаем данные из объекта
12    std::cout << "Employee information obtained by get functions:\n"
13        << fmt::format("name: {}\nsalary: ${:.2f}\n", employee.getName(),
14              employee.getSalary());
15
16    employee.setSalary(500.0); // Изменяем оклад
17    std::cout << "\nUpdated employee information from function toString:\n"
18       << employee.toString();
19
20    // Выводим на экран зарплату сотрудника
21    std::cout << fmt::format("\nearnings: ${:.2f}\n", employee.earnings());
22 }
```

```
Employee information obtained by get functions:
name: Sue Jones
salary: $300.00
```

Окончание ⇨

```
Updated employee information from function toString:
name: Sue Jones
salary: $500.00

earnings: $500.00
```

Илл. 10.3 | Тест-драйв класса SalariedEmployee

10.3.2. Создание иерархии наследования: SalariedEmployee и SalariedCommissionEmployee

Теперь давайте создадим класс `SalariedCommissionEmployee` (илл. 10.4–10.5), унаследованный от `SalariedEmployee` (илл. 10.1–10.2). В этом примере класс `SalariedCommissionEmployee` наследует открытые (`public`) элементы класса `SalariedEmployee`. В дополнение к ним класс `SalariedCommissionEmployee` содержит элементы данных `m_grossSales` и `m_commissionRate` (илл. 10.4, строки 22–23), которые мы перемножим для расчета комиссионного вознаграждения. В строках 13–17 объявляются открытые *get*-функции и *set*-функции, которые оперируют элементами данных `m_grossSales` и `m_commissionRate` класса `SalariedCommissionEmployee`.

```
 1 // Fig. 10.4: SalariedCommissionEmployee.h
 2 // Класс SalariedCommissionEmployee, производный от SalariedEmployee
 3 #pragma once // Предотвращаем повторные включения заголовка
 4 #include <string>
 5 #include <string_view>
 6 #include "SalariedEmployee.h"
 7
 8 class SalariedCommissionEmployee : public SalariedEmployee {
 9 public:
10    SalariedCommissionEmployee(std::string_view name, double salary,
11       double grossSales, double commissionRate);
12
13    void setGrossSales(double grossSales);
14    double getGrossSales() const;
15
16    void setCommissionRate(double commissionRate);
17    double getCommissionRate() const;
18
19    double earnings() const;
20    std::string toString() const;
21 private:
22    double m_grossSales{0.0};
23    double m_commissionRate{0.0};
24 };
```

Илл. 10.4 | Определение класса SalariedCommissionEmployee, унаследованного от SalariedEmployee

Наследование

Двоеточие (:) в строке 8 означает наследование, а слово `public` справа от него означает, что наследование открытое. При открытом наследовании открытые элементы базового класса становятся открытыми элементами производного класса, а защищенные (мы рассмотрим их позже) — защищенными. Класс `SalariedCommissionEmployee` наследует все элементы класса `SalariedEmployee`, кроме конструктора и деструктора. Конструкторы и деструкторы специфичны для каждого класса, поэтому в производных классах должны быть собственные конструкторы и деструкторы[1].

Функции класса SalariedCommissionEmployee

Класс `SalariedCommissionEmployee` содержит открытые функции (открытые сервисы):

- собственный конструктор (строки 10–11);
- `setGrossSales`, `getGrossSales`, `setCommissionRate`, `getCommissionRate`, `earnings` и `toString` (строки 13–20);
- открытые функции, унаследованные от класса `SalariedEmployee`.

Обратите внимание: этих унаследованных функций нет в исходном коде класса `SalariedCommissionEmployee`, тем не менее они являются частями этого класса. Объект `SalariedCommissionEmployee` также содержит скрытые элементы класса `SalariedEmployee`, но они недоступны напрямую в производном классе. Вы можете получить к ним доступ только через унаследованные открытые (или защищенные) функции класса `SalariedEmployee`.

Реализация SalariedCommissionEmployee

На илл. 10.5 показана реализация функций класса `SalariedCommissionEmployee`. Каждый конструктор производного класса должен вызывать конструктор базового класса. Мы делаем это явно с помощью **инициализатора базового класса (base-class initializer)** (строка 11), который передает аргументы конструктору базового класса. В конструкторе класса `SalariedCommissionEmployee` (строки 8–15) в строке 11 вызывается конструктор класса `SalariedEmployee` для инициализации унаследованных элементов данных аргументами `name` и `salary`. Инициализатор базового класса вызывает конструктор базового класса по имени. Функции `setGrossSales` (строки 18–24) и `setCommissionRate` (строки 32–41) проверяют свои аргументы перед изменением элементов `m_grossSales` и `m_commissionRate`.

---

[1] Обычно конструктор производного класса просто передает свои аргументы соответствующему конструктору базового класса без дополнительных действий.

```
// Fig. 10.5: SalariedCommissionEmployee.cpp
// Определения функций класса SalariedCommissionEmployee
#include <fmt/format.h>
#include <stdexcept>
#include "SalariedCommissionEmployee.h"

// Конструктор
SalariedCommissionEmployee::SalariedCommissionEmployee(
   std::string_view name, double salary, double grossSales,
   double commissionRate)
   : SalariedEmployee{name, salary} { // Вызываем конструктор базового класса

   setGrossSales(grossSales); // Объем продаж
   setCommissionRate(commissionRate); // Комиссионная ставка
}

// Записываем объем продаж
void SalariedCommissionEmployee::setGrossSales(double grossSales) {
   if (grossSales < 0.0) {
      throw std::invalid_argument("Gross sales must be >= 0.0");
   }

   m_grossSales = grossSales;
}

// Возвращаем объем продаж
double SalariedCommissionEmployee::getGrossSales() const {
   return m_grossSales;
}

// Возвращаем комиссионную ставку
void SalariedCommissionEmployee::setCommissionRate(
   double commissionRate) {

   if (commissionRate <= 0.0 || commissionRate >= 1.0) {
      throw std::invalid_argument(
         "Commission rate must be > 0.0 and < 1.0");
   }

   m_commissionRate = commissionRate;
}

// Получаем комиссионную ставку
double SalariedCommissionEmployee::getCommissionRate() const {
   return m_commissionRate;
}

// Рассчитываем зарплату с помощью функции SalariedEmployee::earnings()
double SalariedCommissionEmployee::earnings() const {
   return SalariedEmployee::earnings() +
      getGrossSales() * getCommissionRate();
```

```
52 }
53
54 // Возвращаем строковое представление объекта SalariedCommissionEmployee
55 std::string SalariedCommissionEmployee::toString() const {
56    return fmt::format(
57       "{}gross sales: ${:.2f}\ncommission rate: {:.2f}\n",
58       SalariedEmployee::toString(), getGrossSales(), getCommissionRate());
59 }
```

Илл. 10.5 | Определения функций класса SalariedCommissionEmployee

Функция earnings класса SalariedCommissionEmployee

Функция `earnings` класса `SalariedCommissionEmployee` (строки 49–52) переопределяет функцию `earnings` класса `SalariedEmployee` (илл. 10.2, строка 34) для расчета зарплаты сотрудников отдела продаж. В строке 50 в версии класса `SalariedCommissionEmployee` мы используем выражение:

```
SalariedEmployee::earnings()
```

чтобы получить значение зарплаты, основанное на окладе, а затем прибавить к нему комиссионное вознаграждение для расчета итоговой суммы.

Для вызова переопределенной функции базового класса из производного класса надо поместить перед ее именем имя базового класса и оператор разрешения области видимости (`::`). Префикс `SalariedEmployee::` нужен, чтобы избежать бесконечной рекурсии. Кроме того, вызывая функцию класса `SalariedEmployee` из функции класса `SalariedCommissionEmployee`, мы избегаем дублирования кода[1].

Функция toString класса SalariedCommissionEmployee

Аналогично функция `toString` класса `SalariedCommissionEmployee` (илл. 10.5, строки 55–59) переопределяет функцию `toString` класса `SalariedEmployee` (илл. 10.2, строки 37–40). Новая версия возвращает строку, содержащую:

- результат вызова `SalariedEmployee::toString()` (илл. 10.5, строка 58);
- объем продаж и комиссионную ставку.

Тест-драйв класса SalariedCommissionEmployee

На илл. 10.6 показано создание объекта `employee` класса `SalariedCommissionEmployee` (строка 9). В строках 12–16 мы отображаем данные объекта `employee`, вызывая его *get*-функции для получения значений элементов данных. В строках 18–19 используются функции `setGrossSales` и `setCommissionRate` для изменения значений `m_grossSales` и `m_commissionRate` соответственно. Затем в строках 20–21

[1] С одной стороны, отсутствие дубликата кода является преимуществом. С другой стороны, такой подход создает связь между базовым и производными классами, что может затруднить модификацию крупномасштабных систем.

вызывается функция `toString`, чтобы отобразить обновленные данные объекта `employee`. Наконец, в строке 24 отображается обновленная зарплата.

```
1 // fig10_06.cpp
2 // Тест-драйв класса SalariedCommissionEmployee
3 #include <fmt/format.h>
4 #include <iostream>
5 #include "SalariedCommissionEmployee.h"
6
7 int main() {
8    // Создаем объект класса SalariedCommissionEmployee
9    SalariedCommissionEmployee employee{"Bob Lewis", 300.0, 5000.0, .04};
10
11   // Получаем данные из объекта
12   std::cout << "Employee information obtained by get functions:\n"
13      << fmt::format("{}: {}\n{}: {:.2f}\n{}: {:.2f}\n{}: {:.2f}\n",
14         "name", employee.getName(), "salary", employee.getSalary(),
15         "gross sales", employee.getGrossSales(),
16         "commission", employee.getCommissionRate());
17
18   employee.setGrossSales(8000.0); // Изменяем объем продаж
19   employee.setCommissionRate(0.1); // Изменяем комиссионную ставку
20   std::cout << "\nUpdated employee information from function toString:\n"
21      << employee.toString();
22
23   // Выводим на экран зарплату сотрудника
24   std::cout << fmt::format("\nearnings: ${:.2f}\n", employee.earnings());
25 }
```

```
Employee information obtained by get functions:

name: Bob Lewis
salary: $300.00
gross sales: $5000.00
commission: 0.04

Updated employee information from function toString:
name: Bob Lewis
salary: $300.00
gross sales: $8000.00
commission rate: 0.10
earnings: $1100.00
```

Илл. 10.6 | Тест-драйв класса SalariedCommissionEmployee

Конструктор производного класса должен вызывать конструктор базового класса

Компилятор выдаст ошибку, если конструктор класса `SalariedCommissionEmployee` явно не вызывает конструктор класса `SalariedEmployee`. В этом случае компилятор попытается вызвать конструктор класса `SalariedEmployee` по умолчанию, которого не существует, поскольку в базовом классе явно определен пользовательский

конструктор. Если базовый класс предоставляет конструктор по умолчанию, конструктор производного класса может неявно его вызывать.

Дополнительные сведения о конструкторах в производных классах

Конструктор производного класса должен вызывать конструктор базового класса с требуемыми аргументами, иначе возникает ошибка компиляции. Это гарантирует, что унаследованные скрытые элементы базового класса, к которым производный класс не может получить прямой доступ, будут инициализированы. Инициализаторы данных производного класса обычно размещаются в списке инициализаторов после одного или нескольких инициализаторов базового класса.

Из производного класса нет прямого доступа к скрытым элементам базового класса

C++ накладывает строгие ограничения на доступ к скрытым элементам данных. Даже производный класс, тесно связанный со своим базовым классом, не может получить прямой доступ к его скрытым данным. Например, скрытый элемент данных `m_salary` класса `SalariedEmployee` хоть и является частью каждого объекта класса `SalariedCommissionEmployee`, но недоступен напрямую функциям класса `SalariedCommissionEmployee`. При попытке прямого доступа к скрытым данным компилятор выдаст сообщение об ошибке. Например, компилятор GNU g++ сообщит:

```
'double SalariedEmployee::m_salary' is private within this context
```

Однако, как показано на илл. 10.5, функции класса `SalariedCommissionEmployee` могут обращаться напрямую к открытым элементам, унаследованным от класса `SalariedEmployee`.

Заголовок базового класса в заголовке производного класса

Обратите внимание, что мы включили заголовок базового класса в заголовок производного класса (илл. 10.4, строка 6). Это необходимо по следующим причинам:

- Для того чтобы выполнить наследование (илл. 10.4, строка 8), компилятору требуется определение базового класса из `SalariedEmployee.h`.
- Компилятор рассчитывает размер объекта по определению класса. Чтобы создать объект, компилятор должен знать определение класса для резервирования необходимого объема памяти. Размер объекта производного класса зависит от элементов данных, явно объявленных в определении класса, а также от элементов данных, унаследованных от прямых и косвенных базовых классов[1]. Включение определения базового класса позволяет компилятору зарезервировать память с учетом всех элементов данных, влияющих на итоговый размер объекта.

[1] Все объекты класса совместно используют одну копию функций класса, которая хранится отдельно от объектов и не учитывается при расчете их размера.

- Определение базового класса позволяет компилятору проверить правильность использования производным классом унаследованных элементов базового класса. Например, компилятор использует прототипы функций базового класса для проверки вызовов унаследованных функций базового класса из функций производного класса.

Наследование реализации предотвращает повторы кода

При наследовании реализации базовый класс объявляет элементы данных и функции, общие для всех классов иерархии. Когда надо изменить какой-то из этих общих элементов, вы вносите изменения только в базовый класс. Затем производные классы наследуют эти изменения и должны быть перекомпилированы. Без наследования вам пришлось бы вносить изменения во все файлы исходного кода, содержащие одинаковые фрагменты.

10.4. Конструкторы и деструкторы в производных классах

Порядок вызовов конструкторов

Когда создается объект производного класса, начинается цепочка вызовов конструкторов. Перед выполнением своих собственных задач конструктор производного класса вызывает конструктор базового класса: или явно, через инициализатор элемента базового класса, или неявно, вызывая конструктор базового класса по умолчанию. Последний конструктор в этой цепочке — конструктор класса на вершине иерархии. Он завершает выполнение первым, а конструктор самого нижнего производного класса завершает выполнение последним.

Каждый конструктор базового класса инициализирует элементы данных базового класса, наследуемые производными классами. В иерархии классов `SalariedEmployee` — `SalariedCommissionEmployee`, которую мы рассмотрели ранее, при создании объекта `SalariedCommissionEmployee`:

- конструктор этого объекта сразу вызывает конструктор базового класса `SalariedEmployee`;
- класс `SalariedEmployee` является базовым классом в этой иерархии, поэтому выполняется его конструктор, инициализирующий унаследованные объектом `SalariedCommissionEmployee` элементы данных `m_name` и `m_salary`;
- конструктор класса `SalariedEmployee` возвращает управление конструктору объекта класса `SalariedCommissionEmployee` для инициализации элементов данных `m_grossSales` и `m_commissionRate`.

Порядок вызовов деструкторов

Когда уничтожается объект производного класса, программа вызывает деструктор этого объекта, начиная цепочку вызовов деструкторов. Деструкторы выполняются в обратном порядке по сравнению с конструкторами. Когда вызывается деструктор

объекта производного класса, он выполняется и вызывает деструктор следующего класса в иерархии. Это повторяется, пока не будет вызван деструктор базового класса на вершине иерархии.

Конструкторы и деструкторы составных объектов

Предположим, мы создаем объект производного класса, причем базовый и производный классы содержат объекты других классов. Когда создается объект производного класса:

- выполняются конструкторы для объектов-элементов базового класса в том порядке, в каком они объявлены;
- затем выполняется конструктор базового класса;
- затем выполняются конструкторы для объектов-элементов производного класса в том порядке, в каком эти объекты объявлены в производном классе;
- затем выполняется конструктор производного класса.

Деструкторы для элементов данных вызываются в обратном порядке по сравнению с соответствующими конструкторами.

10.5. Принцип динамического полиморфизма на примере объектов в видеоигре

Предположим, мы пишем видеоигру со следующими производными классами: `Martian`, `Venusian`, `Plutonian`, `SpaceShip` и `LaserBeam`. Каждый из этих производных классов наследуется от базового класса `SpaceObject` с функцией `draw` и реализует эту функцию по-своему.

Менеджер экрана

Программа, управляющая выводом на экран (менеджер экрана), работает с вектором (`vector`) указателей `SpaceObject`, указывающих на объекты разных классов. Чтобы обновить изображение, менеджер экрана периодически отправляет каждому объекту одно и то же сообщение через функцию `draw`. Каждый объект реагирует по-своему, например:

- `Martian` рисует себя в виде красного существа с антеннами;
- `SpaceShip` рисует себя в виде серебристой летающей тарелки;
- `LaserBeam` рисует себя в виде ярко-красного луча.

Одно и то же сообщение `draw` вызывает разные формы поведения — это и есть **полиморфизм (polymorphism)**[1].

[1] Это пример динамического полиморфизма (dynamic polymorphism), который также называется полиморфизмом времени выполнения (runtime polymorphism), так как выбор поведения происходит во время выполнения программы. — *Примеч. ред.*

Добавление новых классов в систему

Полиморфный менеджер экрана позволяет легко добавлять в программу новые классы с минимальными изменениями кода. Предположим, вы хотите добавить в игру объекты `Mercurian`. Вы создаете класс `Mercurian`, который наследуется от `SpaceObject` и определяет собственную функцию `draw`, а затем добавляете адрес объекта этого класса в вектор указателей `SpaceObject`. Код менеджера экрана вызывает функцию `draw` одинаково для всех перечисленных в векторе объектов, независимо от их типа. Таким образом, новые объекты `Mercurian` просто «подключаются» к программе.

Благодаря динамическому полиморфизму, вы можете практически без изменений в программе (кроме создания объектов новых классов) добавлять классы по мере необходимости, в том числе те, которые вы не предусмотрели заранее, при разработке программы.

Благодаря динамическому полиморфизму, вы можете работать с общими понятиями, а C++ будет заботиться о конкретных деталях. Вы можете обеспечить правильное поведение объектов, даже не зная их типов, если они принадлежат к одной иерархии классов и обращения к ним выполняются через ссылки или указатели их базовых классов[1].

Динамический полиморфизм способствует расширяемости программ. Полиморфный код создается независимо от конкретных типов объектов, которым вы отправляете сообщения. Таким образом, новые типы объектов, способные обрабатывать существующие сообщения, просто «подключаются» к программе без изменения основной части кода. Для использования новых типов вам надо только модифицировать клиентский код, создающий новые объекты.

10.6. Отношения между объектами в иерархии наследования

В разделе 10.3 мы создали иерархию классов `SalariedEmployee` – `SalariedCommissionEmployee`. Давайте подробнее рассмотрим отношения, существующие в иерархии наследования. На нескольких примерах мы покажем, как направлять указатели базовых и производных классов на объекты базовых и производных классов, а затем вызывать через эти указатели функции для операций с объектами:

- В разделе 10.6.1 мы присвоим адрес объекта производного класса указателю базового класса, а затем покажем, что при вызове функции через указатель базового класса вызывается функциональность базового класса в объекте производного класса. Тип дескриптора определяет, какая из функций будет вызвана.
- В разделе 10.6.2 мы присвоим адрес объекта базового класса указателю производного класса, чтобы вызвать ошибку компиляции. Мы обсудим сообще-

[1] Далее мы рассмотрим такие формы динамического полиморфизма и статического полиморфизма, которые не зависят от иерархий наследования.

ние об ошибке и объясним, почему компилятор не разрешает такие присваивания.

- В разделе 10.6.3 мы присвоим адрес объекта производного класса указателю базового класса, а затем убедимся, что через указатель базового класса можно вызывать только функции, существующие в базовом классе. При попытке вызвать через указатель базового класса функцию, которая есть только в производном классе, возникнет ошибка компиляции.
- В разделе 10.7 мы представим вам **виртуальные функции (virtual functions)**, чтобы показать, как динамический полиморфизм реализуется с помощью указателей базового класса, направленных на объекты производного класса. Мы присвоим адрес объекта производного класса указателю базового класса и вызовем через этот указатель функции производного класса — это и есть полиморфное поведение.

Перечисленные примеры демонстрируют открытое (`public`) наследование, при котором объект производного класса может также рассматриваться и как объект базового класса. В результате вы получаете много интересных возможностей. Например, вы можете создать вектор указателей базового класса, которые указывают на объекты производных классов многих разных типов. Компилятор допускает это, потому что каждый объект производного класса является также и объектом своего базового класса.

С другой стороны, нельзя рассматривать объект базового класса как объект производного класса. Например, объект класса `SalariedEmployee` не является объектом класса `SalariedCommissionEmployee`. В классе `SalariedEmployee` нет элементов данных `m_grossSales`, `m_commissionRate` и соответствующих *set*-функций и *get*-функций. Отношение «является» применимо только в одну сторону: от производного класса к его прямому базовому классу и косвенным базовым классам.

10.6.1. Вызов функций базового класса из объектов производного класса

В программе, показанной на илл. 10.7, мы повторно используем классы `SalariedEmployee` и `SalariedCommissionEmployee` из разделов 10.3.1–10.3.2. Этот пример демонстрирует присваивание указателей базового и производного классов объектам базового и производного классов. Первые два случая логичны и просты:

- мы направляем указатель базового класса на объект базового класса и вызываем функциональность базового класса;
- мы направляем указатель производного класса на объект производного класса и вызываем функциональность производного класса.

Затем мы демонстрируем отношение «является» между производным и базовым классами, направляя указатель базового класса на объект производного класса и показывая, что функциональность базового класса доступна в объекте производного класса.

```
// fig10_07.cpp
// Присваивание адресов объектов базового и производного классов
// указателям базового и производного классов
#include <fmt/format.h>
#include <iostream>
#include "SalariedEmployee.h"
#include "SalariedCommissionEmployee.h"

int main() {
   // Создаем объект базового класса
   SalariedEmployee salaried{"Sue Jones", 500.0};

   // Создаем объект производного класса
   SalariedCommissionEmployee salariedCommission{
      "Bob Lewis", 300.0, 5000.0, .04};

   // Выводим на экран объекты salaried и salariedCommission
   std::cout << fmt::format("{}:\n{}\n{}\n",
      "DISPLAY BASE-CLASS AND DERIVED-CLASS OBJECTS",
      salaried.toString(), // Функция toString базового класса
      salariedCommission.toString()); // Функция toString производного класса

   // Направляем указатель базового класса на объект базового класса
   SalariedEmployee* salariedPtr{&salaried};
   std::cout << fmt::format("{}\n{}:\n{}\n",
      "CALLING TOSTRING WITH BASE-CLASS POINTER TO",
      "BASE-CLASS OBJECT INVOKES BASE-CLASS FUNCTIONALITY",
      salariedPtr->toString()); // Версия из базового класса

   // Направляем указатель производного класса на объект производного класса
   SalariedCommissionEmployee* salariedCommissionPtr{&salariedCommission};

   std::cout << fmt::format("{}\n{}:\n{}\n",
      "CALLING TOSTRING WITH DERIVED-CLASS POINTER TO",
      "DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS FUNCTIONALITY",
      salariedCommissionPtr->toString()); // Версия из производного класса

   // Направляем указатель базового класса на объект производного класса
   salariedPtr = &salariedCommission;
   std::cout << fmt::format("{}\n{}:\n{}\n",
      "CALLING TOSTRING WITH BASE-CLASS POINTER TO DERIVED-CLASS",
      "OBJECT INVOKES BASE-CLASS FUNCTIONALITY",
      salariedPtr->toString()); // Версия из базового класса
}
```

```
DISPLAY BASE-CLASS AND DERIVED-CLASS OBJECTS:
name: Sue Jones
salary: $500.00

name: Bob Lewis
salary: $300.00
gross sales: $5000.00
```

```
commission rate: 0.04

CALLING TOSTRING WITH BASE-CLASS POINTER TO
BASE-CLASS OBJECT INVOKES BASE-CLASS FUNCTIONALITY:
name: Sue Jones
salary: $500.00

CALLING TOSTRING WITH DERIVED-CLASS POINTER TO
DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS FUNCTIONALITY:
name: Bob Lewis
salary: $300.00
gross sales: $5000.00
commission rate: 0.04

CALLING TOSTRING WITH BASE-CLASS POINTER TO DERIVED-CLASS
OBJECT INVOKES BASE-CLASS FUNCTIONALITY:
name: Bob Lewis
salary: $300.00
```

Илл. 10.7 | Присваивание адресов объектов базового и производного классов указателям базового и производного классов

Напомним, что объект `SalariedCommissionEmployee` является объектом `SalariedEmployee`, но с добавлением комиссионного вознаграждения за продажи. Функция `earnings` класса `SalariedCommissionEmployee` (илл. 10.5, строки 49–52) **переопределяет (redefines)** версию класса `SalariedEmployee` (илл. 10.2, строка 34), чтобы прибавить к окладу комиссионное вознаграждение. Функция `toString` класса `SalariedCommissionEmployee` (илл. 10.5, строки 55–59) переопределяет версию класса `SalariedEmployee` (илл. 10.2, строки 37–40), возвращая ту же информацию и комиссионное вознаграждение сотрудника.

Создание объектов и отображение их данных

На илл. 10.7 в строке 11 создается объект `SalariedEmployee`, а в строках 14–15 создается объект `SalariedCommissionEmployee`. В строках 20 и 21 имя каждого объекта используется для вызова его функции `toString`.

Направление указателя базового класса на объект базового класса

В строке 24 указатель `salariedPtr` инициализируется адресом объекта базового класса `salaried`. В строке 28 этот указатель используется для вызова функции `toString` объекта `salaried` из базового класса `SalariedEmployee`.

Направление указателя производного класса на объект производного класса

В строке 31 указатель `salariedCommissionPtr` инициализируется адресом объекта производного класса `salariedCommission`. В строке 36 этот указатель используется для вызова функции `toString` объекта `salariedCommission` из производного класса `SalariedCommissionEmployee`.

Направление указателя базового класса на объект производного класса

В строке 39 адрес объекта производного класса `salariedCommission` присваивается указателю базового класса `salariedPtr`. Такое «пересечение» разрешено, потому что объект производного класса является также и объектом своего базового класса. В строке 43 этот указатель используется для вызова функции `toString`. Несмотря на то что указатель базового класса указывает на объект производного класса, вызывается функция базового класса. Объем продаж и комиссионная ставка не отображаются, поскольку они не являются элементами данных базового класса.

Выходные данные этой программы показывают, что выбор вызываемой функции зависит от типа указателя (или от типа ссылки, как вы скоро увидите), через который она вызывается, а не от типа объекта, для которого она вызывается. В разделе 10.7 мы покажем, что виртуальные функции позволяют выбирать нужную функциональность в зависимости от типа объекта, — это важный аспект реализации динамического полиморфного поведения.

10.6.2. Направление указателя производного класса на объект базового класса

Теперь давайте попробуем направить указатель производного класса на объект базового класса (илл. 10.8). В строке 7 мы создаем объект класса `SalariedEmployee`. В строке 11 мы инициализируем указатель `SalariedCommissionEmployee` адресом объекта `salaried` базового класса. Компилятор сообщает об ошибке, поскольку `SalariedEmployee` не является `SalariedCommissionEmployee`.

```
1 // fig10_08.cpp
2 // Присваивание адреса объекта базового класса указателю производного класса!
3 #include "SalariedEmployee.h"
4 #include "SalariedCommissionEmployee.h"
5
6 int main() {
7    SalariedEmployee salaried{"Sue Jones", 500.0};
8
9    // Направляем указатель производного класса на объект базового класса
10   // Ошибка: SalariedEmployee не является SalariedCommissionEmployee!
11   SalariedCommissionEmployee* salariedCommissionPtr{&salaried};
12 }
```

Сообщение компилятора Microsoft Visual C++ об ошибке:

```
fig10_08.cpp(11,63): error C2440: 'initializing': cannot convert from
'SalariedEmployee *' to 'SalariedCommissionEmployee *'
```

Илл. 10.8 | Ошибочное присваивание адреса объекта базового класса указателю производного класса

Подумайте, что было бы, если бы компилятор разрешил такое присваивание. Через указатель `SalariedCommissionEmployee` можно вызвать любую функцию этого

класса (в том числе `setGrossSales` и `setCommissionRate`) для объекта, на который направлен указатель, то есть для объекта `salaried` базового класса. Но в объекте `SalariedEmployee` нет ни функций `setGrossSales` и `setCommissionRate`, ни элементов данных `m_grossSales` и `m_commissionRate`. Из-за такого присваивания возникли бы проблемы, поскольку функции `setGrossSales` и `setCommissionRate` предполагали бы, что элементы данных `m_grossSales` и `m_commissionRate` расположены «на своих обычных местах» внутри объекта `SalariedCommissionEmployee`. Но в памяти объекта `SalariedEmployee` нет этих элементов данных, поэтому функции `setGrossSales` и `setCommissionRate` повредили бы другие данные в памяти.

10.6.3. Вызовы функций производного класса через указатель базового класса

Компилятор позволяет вызывать через указатель базового класса только функции базового класса. Если указатель базового класса указывает на объект производного класса, то при обращении к функции, существующей только в производном классе, возникает ошибка компиляции. На илл. 10.9 показаны сообщения компилятора при вызове функции производного класса через указатель базового класса (строки 22–24).

```
1 // fig10_09.cpp
2 // Вызовы функций, существующих только в производном классе,
3 // через указатель базового класса!
4 #include <string>
5 #include "SalariedEmployee.h"
6 #include "SalariedCommissionEmployee.h"
7
8 int main() {
9    SalariedCommissionEmployee salariedCommission{
10      "Bob Lewis", 300.0, 5000.0, .04};
11
12   // Направляем указатель базового класса на объект производного класса
13   SalariedEmployee* salariedPtr{&salariedCommission};
14
15   // Вызываем функции базового класса для объекта производного класса
16   // через указатель базового класса
17   std::string name{salariedPtr->getName()};
18   double salary{salariedPtr->getSalary()};
19
20   // Вызываем функции, существующие только в производном классе, для объекта
21   // производного класса через указатель базового класса (запрещено!)
22   double grossSales{salariedPtr->getGrossSales()};
23   double commissionRate{salariedPtr->getCommissionRate()};
24   salariedPtr->setGrossSales(8000.0);
25 }
```

Сообщения компилятора GNU C++ об ошибках:

```
fig10_09.cpp: In function 'int main()':
```

Окончание ⇨

```
fig10_09.cpp:22:35: error: 'class SalariedEmployee' has no member named
'getGrossSales'
   22 | double grossSales{salariedPtr->getGrossSales()};
      |                                  ^~~~~~~~~~~~~
fig10_09.cpp:23:39: error: 'class SalariedEmployee' has no member named
'getCommissionRate'
   23 | double commissionRate{salariedPtr->getCommissionRate()};
      |                                    ^~~~~~~~~~~~~~~~~~
fig10_09.cpp:24:17: error: 'class SalariedEmployee' has no member named
'setGrossSales'
   24 | salariedPtr->setGrossSales(8000.0);
      |              ^~~~~~~~~~~~~
```

Илл. 10.9 | Ошибочные вызовы функций, существующих только в производном классе, через указатель базового класса

В строках 9–10 создается объект класса `SalariedCommissionEmployee`. В строке 13 указатель базового класса `salariedPtr` инициализируется адресом объекта производного класса `salariedCommission`. Это разрешено, потому что `SalariedCommissionEmployee` является `SalariedEmployee`.

В строках 17–18 функции базового класса вызываются через указатель базового класса. Эти вызовы разрешены, потому что `SalariedCommissionEmployee` наследует эти функции.

Мы знаем, что `salariedPtr` указывает на объект `SalariedCommissionEmployee`. В строках 22–24 мы пробуем вызывать три функции, которые есть только в `SalariedCommissionEmployee`: `getGrossSales`, `getCommissionRate` и `setGrossSales`. Компилятор сообщает об ошибках, поскольку этих функций нет в базовом классе `SalariedEmployee`. Через указатель `salariedPtr` можно вызывать только функции базового класса `SalariedEmployee`.

10.7. Виртуальные функции и виртуальные деструкторы

В разделе 10.6.1 мы создали указатель базового класса `SalariedEmployee`, указывающий на объект производного класса `SalariedCommissionEmployee`, и использовали его для вызова функции `toString`. В этом случае была вызвана функция класса `SalariedEmployee`. Как же вызвать функцию `toString` производного класса через указатель базового класса?

10.7.1. Зачем нужны виртуальные функции

Предположим, что классы `Circle`, `Triangle`, `Rectangle` и `Square` — производные от базового класса `Shape`. Они рисуют на экране свои объекты с помощью функции `draw`, но ее реализация в каждом классе немного отличается. В программе, которая выводит на экран много разных фигур, удобно рассматривать все фигуры оди-

наково: как объекты базового класса `Shape`. Тогда мы сможем нарисовать любую фигуру, просто вызвав функцию `draw` через указатель базового класса. Во время выполнения программа сама выберет нужную функцию `draw` (функцию `draw` из соответствующего производного класса), основываясь на типе объекта, на который направлен указатель базового класса. Это динамическое полиморфное поведение. При вызове виртуальной (`virtual`) функции автоматически выбирается функция соответствующего класса в зависимости от типа объекта, к которому происходит обращение по указателю (или по ссылке), а не от типа этого указателя (или ссылки).

10.7.2. Объявление виртуальной функции

Чтобы реализовать динамическое полиморфное поведение, мы объявляем функцию базового класса `draw` виртуальной (`virtual`)[1], а затем переопределяем ее в каждом производном классе для отображения соответствующей фигуры. Переопределенная функция в производном классе должна иметь ту же сигнатуру, что и функция в базовом классе. Чтобы объявить виртуальную функцию, перед ее прототипом добавляется ключевое слово `virtual`. Пример прототипа функции `draw` в базовом классе `Shape`:

```
virtual void draw() const;
```

Этот прототип объявляет, что `draw` — виртуальная функция, которая не принимает аргументы и ничего не возвращает. Функция `draw` объявлена как `const`, потому что она не должна изменять объект `Shape`, для которого вызывается. Виртуальные функции не обязательно должны объявляться как `const`, они могут получать аргументы и возвращать значения при необходимости. Если функция объявлена виртуальной, она становится виртуальной во всех классах, прямо или косвенно производных от данного базового класса. Если производный класс не переопределяет виртуальную функцию базового класса, он просто наследует реализацию виртуальной функции от своего базового класса.

10.7.3. Вызов виртуальной функции

Если виртуальная функция вызывается

- через указатель базового класса на объект производного класса (например, `shapePtr->draw()`)
- или через ссылку базового класса на объект производного класса (например, `shapeRef.draw()`),

то программа выберет во время выполнения соответствующую функцию производного класса, основываясь на типе объекта, а не на типе указателя или ссылки.

[1] В некоторых языках программирования, в том числе Java и Python, все функции (методы) работают подобно виртуальным функциям C++. В разделе 10.10 мы покажем, что виртуальные функции немного снижают быстродействие и увеличивают потребление памяти. C++ позволяет вам выбирать, какие функции должны быть виртуальными, исходя из требований к быстродействию ваших приложений.

Если выбор нужной функции происходит во время выполнения программы, это называется **динамическим связыванием (dynamic binding)**, или **поздним связыванием (late binding)**.

Когда виртуальная функция вызывается через имя объекта с оператором «точка» (например, `squareObject.draw()`), оптимизирующий компилятор может выбрать функцию заранее, на этапе компиляции. Это называется **статическим связыванием (static binding)**. Виртуальная функция, которая будет вызвана, — это функция, определенная для класса этого объекта.

10.7.4. Виртуальные функции в иерархии SalariedEmployee

Давайте рассмотрим, как виртуальные функции могут обеспечить полиморфизм в нашей иерархии. Мы внесли только два изменения в заголовок каждого класса, чтобы получить полиморфное поведение, которое вы увидите на илл. 10.10. В заголовке класса `SalariedEmployee` (илл. 10.1) мы изменили прототипы функций `earnings` и `toString` (строки 17–18):

```
double earnings() const;
std::string toString() const;
```

Теперь к ним добавлено ключевое слово `virtual`:

```
virtual double earnings() const;
virtual std::string toString() const;
```

В заголовке класса `SalariedCommissionEmployee` (илл. 10.4) мы изменили прототипы функций `earnings` и `toString` (строки 19–20):

```
double earnings() const;
std::string toString() const;
```

Теперь к ним добавлено ключевое слово `override` (обсуждается после илл. 10.10):

```
double earnings() const override;
std::string toString() const override;
```

Функции `earnings` и `toString` класса `SalariedEmployee` являются виртуальными, поэтому их версии из производного класса `SalariedCommissionEmployee` переопределяют версии из класса `SalariedEmployee`. Мы не меняем реализации функций в классах `SalariedEmployee` и `SalariedCommissionEmployee`. Мы повторно используем версии, показанные на илл. 10.2 и 10.5.

Динамическое полиморфное поведение

Теперь, если мы присвоим указатель базового класса `SalariedEmployee` объекту производного класса `SalariedCommissionEmployee` и используем этот указатель для вызова функций `earnings` или `toString`, эти функции объектов производного класса будут вызваны полиморфно. Код на илл. 10.10 демонстрирует это полиморфное поведение. Сначала в строках 10–21 создаются объекты классов `SalariedEmployee` и `SalariedCommissionEmployee`, затем имена этих объектов ис-

пользуются для отображения результатов вызова функции `toString`. Это поможет подтвердить результаты динамического связывания далее в программе. В строках 27–38 снова показывается, что:

- указатель `SalariedEmployee`, направленный на объект `SalariedEmployee`, может использоваться для вызова функциональности `SalariedEmployee`;
- указатель `SalariedCommissionEmployee`, направленный на объект `SalariedCommissionEmployee`, может использоваться для вызова функциональности `SalariedCommissionEmployee`.

В строке 41 указатель базового класса `salariedPtr` направляется на объект производного класса `salariedCommission`. В строке 48 вызывается функция `toString` через указатель базового класса. Как вы можете видеть в выходных данных, вызывается функция `toString` объекта производного класса `salariedCommission`. Объявление функции класса как виртуальной и ее вызов через указатель или ссылку базового класса приводит к тому, что программа во время выполнения выбирает вызываемую функцию в зависимости от типа объекта.

```
1 // fig10_10.cpp
2 // Полиморфизм, виртуальные функции и динамическое связывание
3 #include <fmt/format.h>
4 #include <iostream>
5 #include "SalariedEmployee.h"
6 #include "SalariedCommissionEmployee.h"
7
8 int main() {
9    // Создаем объект базового класса
10   SalariedEmployee salaried{"Sue Jones", 500.0};
11
12   // Создаем объект производного класса
13   SalariedCommissionEmployee salariedCommission{
14      "Bob Lewis", 300.0, 5000.0, .04};
15
16   // Выводим данные, используя статическое связывание
17   std::cout << fmt::format("{}\n{}:\n{}\n{}\n",
18      "INVOKING TOSTRING FUNCTION ON BASE-CLASS AND",
19      "DERIVED-CLASS OBJECTS WITH STATIC BINDING",
20      salaried.toString(), // Статическое связывание
21      salariedCommission.toString()); // Статическое связывание
22
23   std::cout << "INVOKING TOSTRING FUNCTION ON BASE-CLASS AND\n"
24      << "DERIVED-CLASS OBJECTS WITH DYNAMIC BINDING\n\n";
25
26   // Направляем указатель базового класса на объект базового класса
27   SalariedEmployee* salariedPtr{&salaried};
28   std::cout << fmt::format("{}\n{}:\n{}\n",
29      "CALLING VIRTUAL FUNCTION TOSTRING WITH BASE-CLASS POINTER",
30      "TO BASE-CLASS OBJECT INVOKES BASE-CLASS FUNCTIONALITY",
31      salariedPtr->toString()); // Версия из базового класса
32
```

Окончание ⇨

```
33    // Направляем указатель производного класса на объект производного класса
34    SalariedCommissionEmployee* salariedCommissionPtr{&salariedCommission};
35    std::cout << fmt::format("{}\n{}:\n{}\n",
36       "CALLING VIRTUAL FUNCTION TOSTRING WITH DERIVED-CLASS POINTER",
37       "TO DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS FUNCTIONALITY",
38       salariedCommissionPtr->toString()); // Версия из производного класса
39
40    // Направляем указатель базового класса на объект производного класса
41    salariedPtr = &salariedCommission;
42
43    // Динамический полиморфизм: вызываем SalariedCommissionEmployee
44    // через указатель базового класса на объект производного класса
45    std::cout << fmt::format("{}\n{}:\n{}\n",
46       "CALLING VIRTUAL FUNCTION TOSTRING WITH BASE-CLASS POINTER",
47       "TO DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS FUNCTIONALITY",
48       salariedPtr->toString()); // Версия из производного класса
49 }
```

```
INVOKING TOSTRING FUNCTION ON BASE-CLASS AND
DERIVED-CLASS OBJECTS WITH STATIC BINDING:
name: Sue Jones
salary: $500.00

name: Bob Lewis
salary: $300.00
gross sales: $5000.00
commission rate: 0.04

INVOKING TOSTRING FUNCTION ON BASE-CLASS AND
DERIVED-CLASS OBJECTS WITH DYNAMIC BINDING

CALLING VIRTUAL FUNCTION TOSTRING WITH BASE-CLASS POINTER
TO BASE-CLASS OBJECT INVOKES BASE-CLASS FUNCTIONALITY:
name: Sue Jones
salary: $500.00

CALLING VIRTUAL FUNCTION TOSTRING WITH DERIVED-CLASS POINTER
TO DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS FUNCTIONALITY:
name: Bob Lewis
salary: $300.00
gross sales: $5000.00
commission rate: 0.04

CALLING VIRTUAL FUNCTION TOSTRING WITH BASE-CLASS POINTER
TO DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS FUNCTIONALITY:
name: Bob Lewis
salary: $300.00
gross sales: $5000.00
commission rate: 0.04
```

Илл. 10.10 | Полиморфизм: вызов виртуальной функции производного класса через указатель базового класса на объект производного класса

Когда `salariedPtr` указывает на объект `SalariedEmployee`, вызывается функция `toString` класса `SalariedEmployee` (строка 31). Когда `salariedPtr` указывает на объект `SalariedCommissionEmployee`, вызывается функция `toString` класса `SalariedCommissionEmployee` (строка 48). Таким образом, один и тот же вызов функции `toString` через указатели базового класса на различные объекты принимает много форм (в данном случае две формы). Это и есть полиморфизм.

Виртуальные функции не следует вызывать из конструкторов и деструкторов

При вызове виртуальной функции из конструктора или деструктора базового класса выбирается ее версия из базового класса, даже если конструктор или деструктор базового класса вызывается при создании или уничтожении объекта производного класса. Это не то поведение, которое вы ожидаете от виртуальных функций, поэтому в C++ Core Guidelines рекомендуется не вызывать виртуальные функции из конструкторов и деструкторов[1].

Ключевое слово override (C++11)

Объявление функций `earnings` и `toString` класса `SalariedCommissionEmployee` с ключевым словом `override` требует от компилятора проверить наличие в базовом классе виртуальной функции с идентичной сигнатурой. Если ее там нет, компилятор выдает сообщение об ошибке. Это гарантирует, что вы переопределите соответствующую функцию базового класса. Это также предотвращает случайное сокрытие функции базового класса с тем же именем, но с другой сигнатурой. Итак, чтобы избежать ошибок, добавляйте ключевое слово `override` в прототип каждой функции производного класса, переопределяющей виртуальную функцию базового класса.

В C++ Core Guidelines поясняется, что:

- ключевое слово `virtual` определяет новую виртуальную функцию в иерархии;
- ключевое слово `override` указывает, что функция производного класса переопределяет виртуальную функцию базового класса.

Поэтому вы должны использовать `virtual` или `override` в прототипе каждой виртуальной функции[2].

---

[1] «C.82: Don't Call Virtual Functions in Constructors and Destructors». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-ctor-virtual.

[2] «C.128: Virtual Functions Should Specify Exactly One of virtual, override, or final». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rhoverride.

10.7.5. Виртуальные деструкторы

В C++ Core Guidelines рекомендуется включать виртуальный деструктор в каждый класс, содержащий виртуальные функции[1]. Это помогает предотвращать трудно выявляемые ошибки в тех случаях, когда в производном классе есть пользовательский деструктор. Если в классе нет деструктора, компилятор генерирует его автоматически, но этот деструктор не будет виртуальным. Поэтому в современном C++ определение виртуального деструктора для большинства классов выглядит примерно так:

```
virtual ~SalariedEmployee() = default;
```

При такой форме записи деструктор объявляется виртуальным, но компилятор все равно генерирует для класса деструктор по умолчанию (`= default`).

10.7.6. Финальные функции и классы

До C++11 производный класс мог переопределять любую виртуальную функцию базового класса. Начиная с C++11, виртуальную функцию можно объявить в прототипе как финальную (`final`):

возвращаемыйТип имяФункции(параметры) `final`;

Финальная функция не может быть переопределена ни в одном производном классе. В многоуровневой иерархии классов это гарантирует, что определение такой функции будет использоваться всеми последующими прямыми и косвенными производными классами.

До C++11 любой класс мог стать базовым классом в любой точке иерархии. В C++11 и более поздних версиях можно объявить класс финальным (`final`), чтобы запретить его использование в качестве базового класса:

```
class MyClass final {
// Тело класса
};
```

или:

```
class DerivedClass : public BaseClass final {
// Тело класса
};
```

Попытка переопределить финальную функцию или выполнить наследование от финального класса приведет к ошибке компиляции.

Если виртуальная функция объявлена финальной, это дает преимущество в быстродействии. Компилятор оптимизирует исполняемый код, когда знает, что вир-

[1] «C.35: A Base Class Destructor Should Be Either public and virtual, or protected and Non-virtual». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-dtor-virtual.

туальная функция наверняка не будет переопределяться. Например, компилятор может уже во время компиляции выбрать нужную функцию из нескольких[1]. Такая оптимизация называется **девиртуализацией (devirtualization)**[2].

Иногда компилятор девиртуализирует вызовы даже тех виртуальных функций, которые объявлены без спецификатора `final`. Например, компилятор может распознать во время компиляции тип объекта, который будет использоваться во время выполнения. В этом случае уже на этапе компиляции возможна оптимизация вызова соответствующей виртуальной функции.

10.8. Абстрактные классы и чистые виртуальные функции

Иногда требуется определить класс, из которого никогда не будут создаваться объекты. Такой **абстрактный класс (abstract class)** определяет открытый (`public`) интерфейс для производных классов в иерархии. Поскольку абстрактные классы используются в качестве базовых классов в иерархиях наследования, их называют **абстрактными базовыми классами (abstract base classes)**. Их нельзя использовать для создания объектов, потому что абстрактные классы содержат не все необходимые для этого элементы. Недостающие элементы надо определить в производных классах, чтобы получить возможность создания объектов производных классов. Мы рассмотрим программы с абстрактными классами в разделах 10.9, 10.12 и 10.14.

Классы, из которых можно создавать объекты, называются **конкретными классами (concrete classes)**. Они определяют или наследуют реализации каждой функции, определенной в базовом классе. Хороший пример абстрактного класса — базовый класс `Shape` из раздела 10.2. Там же мы показали на диаграмме абстрактный базовый класс `TwoDimensionalShape` с конкретными производными классами `Circle`, `Square` и `Triangle` и абстрактный базовый класс `ThreeDimensionalShape` с конкретными производными классами `Sphere`, `Cube` и `Tetrahedron`. Абстрактные базовые классы являются слишком обобщенными для определения объектов. Например, если кто-то попросит вас «нарисовать двумерную фигуру», какую фигуру вы нарисуете? Конкретные классы предоставляют все необходимые уточнения, позволяющие создавать объекты.

10.8.1. Чистые виртуальные функции

Класс становится абстрактным, если в нем объявлена одна или несколько **чистых виртуальных функций (pure virtual functions)**. Такие функции обозначаются символами «`= 0`» в прототипе, например:

```
virtual void draw() const = 0; // Чистая виртуальная функция
```

[1] Мэтт Годболт (Matt Godbolt), «Optimizations in C++ Compilers». 12 ноября 2019 г. Был доступен: 15 сентября 2023 г. https://queue.acm.org/detail.cfm?id=3372264.

[2] В стандарте C++ описан ряд возможных оптимизаций, но они необязательны. Оптимизация выполняется по усмотрению разработчиков компилятора.

Здесь символы «= 0» называются **спецификатором чистой функции (pure specifier)**. Чистая виртуальная функция не предоставляет реализацию. Каждый конкретный производный класс должен переопределить чистые виртуальные функции базового класса конкретными реализациями. Если производный класс не переопределяет чистую виртуальную функцию своего базового класса, он тоже будет абстрактным. Для сравнения, обычная виртуальная функция имеет реализацию, предоставляющую каждому производному классу выбор: переопределить функцию или наследовать реализацию базового класса. Абстрактный класс может содержать элементы данных и конкретные функции. Как будет показано в разделе 10.12, абстрактный класс, в котором все функции являются чистыми виртуальными, называется **чистым абстрактным классом (pure abstract class)** или **интерфейсом (interface)**.

Чистая виртуальная функция используется в тех случаях, когда реализовать функцию в базовом классе невозможно, а во всех конкретных производных классах — необходимо. В нашем примере с космической игрой не имеет смысла включать в базовый класс `SpaceObject` реализацию функции `draw`. Невозможно нарисовать «просто космический объект», не зная, какой конкретно объект имеется в виду.

Хотя нельзя создавать объекты абстрактного базового класса, можно объявлять указатели и ссылки с типом абстрактного базового класса. Они могут обозначать объекты любых конкретных классов, производных от абстрактного базового класса. Такие указатели и ссылки обычно используются для полиморфного управления объектами производных классов во время выполнения.

10.8.2. Драйверы устройств: полиморфизм в операционных системах

Полиморфизм очень эффективен при реализации многоуровневых программных систем. Например, операционная система должна учитывать, что каждый тип физического устройства ввода-вывода может работать иначе, чем другие. Несмотря на это, сообщения ввода-вывода, передающие данные между операционной системой и устройствами, должны быть одинаковыми для всей группы устройств. Сообщение о выводе данных, отправленное объекту драйвера устройства, должно интерпретироваться в контексте этого конкретного драйвера, с учетом того, как он взаимодействует с устройством. Но само это сообщение («скопировать несколько байтов из памяти в устройство») не отличается от других сообщений операционной системы, выводящих данные в другие устройства.

Рассмотрим объектно-ориентированную операционную систему, в которой есть совместимый со всеми драйверами интерфейс, основанный на абстрактном базовом классе. От этого класса наследуются производные классы, которые выполняют примерно одинаковые задачи. Открытые (`public`) функции, входящие в драйверы устройств, являются чистыми виртуальными функциями абстрактного базового класса. Реализации этих функций находятся в производных классах, соответствующих конкретным драйверам.

Эта архитектура позволяет легко добавлять в операционную систему новые устройства. Пользователь может просто подключить устройство к разъему ком-

пьютера и установить его драйвер. После этого операционная система будет «общаться» с новым устройством через открытые функции драйвера. Эти функции аналогичны функциям других драйверов, так как основаны на абстрактном базовом классе всей группы драйверов.

10.9. Полиморфизм в программе расчета зарплаты

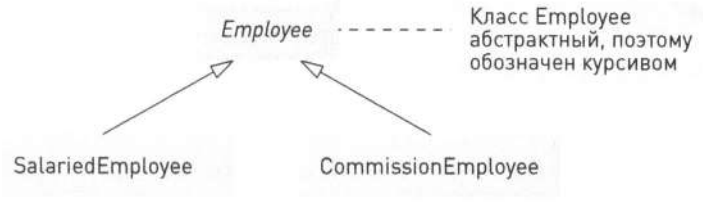
Рассмотрим использование абстрактных классов и полиморфизма для выполнения расчетов зарплаты двух типов сотрудников. Мы создадим простую иерархию сотрудников для решения следующей задачи:

> *Компания еженедельно платит зарплату сотрудникам, которые делятся на две категории:*
>
> 1) *сотрудники, получающие фиксированную зарплату (оклад);*
> 2) *сотрудники, получающие в качестве зарплаты определенный процент от их продаж.*
>
> *Компания хочет внедрить написанную на C++ программу расчета зарплаты.*

Многие иерархии начинаются с абстрактного базового класса, от которого наследуются финальные (`final`) производные классы. Мы будем использовать абстрактный класс `Employee`, который представляет обобщенного сотрудника и определяет «интерфейс» иерархии, то есть набор функций, которые должны быть у всех производных классов (представляющих разные категории сотрудников), чтобы программа могла вызывать эти функции для любых объектов этих классов.

Независимо от способа расчета зарплаты, у каждого сотрудника есть имя. Для хранения имени сотрудника мы определим скрытый элемент данных `m_name` в абстрактном базовом классе `Employee`.

Конечные классы, которые наследуются непосредственно от базового класса `Employee`, — это `SalariedEmployee` и `CommissionEmployee`. Каждый из них является последним звеном в иерархии и не может быть базовым классом. Следующая диаграмма показывает иерархию наследования для нашей программы расчета зарплаты. Имя абстрактного класса `Employee` выделено курсивом, как это принято на таких UML-диаграммах.



Производный класс может наследовать от базового класса интерфейс и (или) реализацию. В тех иерархиях, которые предназначены для **наследования интерфейса (interface inheritance)**, конкретные определения функциональности обычно находятся внизу. На вершине иерархии находится базовый класс, объявляющий одну или несколько функций, которые должны быть определены и реализованы в каждом из производных классов.

В следующих разделах мы реализуем иерархию класса `Employee`. В разделах 10.9.1–10.9.3 определяются все три класса иерархии: абстрактный и два конкретных. В разделе 10.9.4 показана тестовая программа, которая создает объекты конкретных классов и обрабатывает их по принципу динамического полиморфизма.

10.9.1. Создание абстрактного базового класса Employee

Класс `Employee` (илл. 10.11–10.12) предоставляет функции `earnings` и `toString`, а также *get*-функции и *set*-функции, оперирующие элементом данных `m_name`. Функция `earnings`, безусловно, применима в целом ко всем объектам `Employee`, но каждый расчет зарплаты зависит от конкретного класса. Таким образом, мы объявляем `earnings` чистой виртуальной функцией в базовом классе `Employee`, потому что ее реализация по умолчанию не имеет смысла в контексте базового класса, без дополнительной информации, от которой зависит размер зарплаты.

Каждый производный класс должен переопределить функцию `earnings` с подходящей реализацией. При расчете зарплаты сотрудника программа присваивает адрес соответствующего объекта указателю базового класса `Employee`, а затем вызывает функцию `earnings` этого объекта.

Тестовая программа содержит вектор указателей `Employee`. Каждый из них указывает на объект, который является `Employee`, то есть на какой-либо объект конкретного класса, производного от `Employee`. Программа перебирает элементы вектора и вызывает функцию `earnings` для каждого сотрудника. C++ обрабатывает эти вызовы полиморфно. Поскольку `earnings` определена в базовом классе как чистая виртуальная функция, ее должен переопределить каждый производный класс, который «хочет» быть конкретным классом.

Функция `toString` класса `Employee` возвращает строку с именем сотрудника. Каждый класс, производный от `Employee`, переопределяет функцию `toString`, чтобы она возвращала не только имя сотрудника, но и дополнительные сведения о нем. Функция `toString` каждого производного класса может вызывать функцию `earnings`, хотя `earnings` — это чистая виртуальная функция в классе `Employee`. Учитывая, что в каждом конкретном классе гарантированно есть реализация функции `earnings`, ее можно вызывать даже из функции `toString` класса `Employee`. Вызывая функцию `toString` через указатель или через ссылку во время выполнения, вы всегда вызываете ее для конкретного объекта производного класса.

В следующей таблице вы видите соответствие между тремя классами нашей иерархии и функциями `earnings` и `toString`.

	earnings	toString
Employee	pure virtual	name: *m_name*
Salaried-Employee	*m_salary*	name: *m_name* salary: *m_salary*
Commission-Employee	*m_commissionRate* * *m_grossSales*	name: *m_name* gross sales: *m_grossSales* commission rate: *m_commissionRate*

Курсивом показано, какие значения объектов используются в функциях `earnings` и `toString`. Мы отметили, что в классе `Employee` функция `earnings` является `pure virtual`, потому что этот класс не предоставляет ее реализацию. Каждый производный класс переопределяет эту функцию, предоставляя соответствующую реализацию. Мы не перечислили *get*-функции и *set*-функции базового класса `Employee`, поскольку производные классы их не переопределяют. Каждая из этих функций наследуется и используется производными классами в исходном виде.

Заголовок класса Employee

Рассмотрим заголовок класса `Employee` (илл. 10.11). В нем есть открытые функции:

- конструктор, принимающий имя сотрудника в качестве аргумента (строка 9);
- виртуальный деструктор по умолчанию (C++11) (строка 10);
- *set*-функция для изменения имени сотрудника (строка 12);
- *get*-функция для получения имени сотрудника (строка 13);
- чистая виртуальная функция `earnings` (строка 16);
- виртуальная функция `toString` (строка 17).

Напомним, что мы объявляем функцию `earnings` как чистую виртуальную, потому что для определения способа расчета зарплаты надо знать категорию сотрудника. Каждый конкретный производный класс предоставляет реализацию функции `earnings`. Используя указатель или ссылку базового класса `Employee`, программа может вызывать функцию `earnings` для объекта любого конкретного класса, производного от `Employee`.

```
1 // Fig. 10.11: Employee.h
2 // Абстрактный базовый класс Employee
3 #pragma once // Предотвращаем повторные включения заголовка
4 #include <string>
5 #include <string_view>
6
7 class Employee {
8 public:
9    explicit Employee(std::string_view name);
```

Окончание ⇨

```
10    virtual ~Employee() = default; // Компилятор генерирует вирт. деструктор
11
12    void setName(std::string_view name);
13    std::string getName() const;
14
15    // Чистая виртуальная функция делает класс Employee абстрактным базовым
16    virtual double earnings() const = 0; // Чистая виртуальная функция
17    virtual std::string toString() const; // Виртуальная функция
18 private:
19    std::string m_name;
20 };
```

Илл. 10.11 | Абстрактный базовый класс Employee

Определения функций класса Employee

На илл. 10.12 показаны определения функций класса **Employee**. Виртуальная функция **earnings** здесь не реализована, а реализация виртуальной функции **toString** (строки 17–19) будет переопределена в каждом производном классе. Эти переопределенные функции **toString** будут вызывать функцию **toString** базового класса **Employee**, чтобы получать **string** с общей информацией для всех классов в иерархии (с именем сотрудника).

```
 1 // Fig. 10.12: Employee.cpp
 2 // Определения функций абстрактного базового класса Employee
 3 // Внимание: чистые виртуальные функции не определены
 4 #include <fmt/format.h>
 5 #include "Employee.h" // Определение класса Employee
 6
 7 // Конструктор
 8 Employee::Employee(std::string_view name) : m_name{name} {} // Пустое тело
 9
10 // Записываем имя
11 void Employee::setName(std::string_view name) {m_name = name;}
12
13 // Получаем имя
14 std::string Employee::getName() const {return m_name;}
15
16 // Возвращаем строковое представление данных Employee
17 std::string Employee::toString() const {
18    return fmt::format("name: {}", getName());
19 }
```

Илл. 10.12 | Файл реализации класса Employee

10.9.2. Создание конкретного производного класса SalariedEmployee

Класс **SalariedEmployee** (илл. 10.13–10.14) является производным от класса **Employee** (илл. 10.13, строка 8). В классе **SalariedEmployee** есть открытые функции:

- конструктор, принимающий имя сотрудника и оклад в качестве аргументов (строка 10);
- виртуальный деструктор по умолчанию (C++11) (строка 11);
- *set*-функция для записи нового неотрицательного значения в элемент данных `m_salary` (строка 13) и *get*-функция для возврата значения `m_salary` (строка 14);
- переопределенная виртуальная функция `earnings`, вычисляющая зарплату для объекта `SalariedEmployee` (строка 17);
- переопределенная виртуальная функция `toString` (строка 18), возвращающая строковое представление объекта `SalariedEmployee`.

```
1 // Fig. 10.13: SalariedEmployee.h
2 // Класс SalariedEmployee, производный от Employee
3 #pragma once
4 #include <string> // Класс string стандартной библиотеки C++
5 #include <string_view>
6 #include "Employee.h" // Определение класса Employee
7
8 class SalariedEmployee final : public Employee {
9 public:
10    SalariedEmployee(std::string_view name, double salary);
11    virtual ~SalariedEmployee() = default; // Виртуальный деструктор
12
13    void setSalary(double salary);
14    double getSalary() const;
15
16    // Ключевое слово override указывает на переопределение
17    double earnings() const override; // Расчет зарплаты
18    std::string toString() const override; // Строковое представление
19 private:
20    double m_salary{0.0};
21 };
```

Илл. 10.13 | Заголовок класса SalariedEmployee

Определения функций класса SalariedEmployee

На илл. 10.14 показаны функции класса `SalariedEmployee`:

- Конструктор передает аргумент `name` конструктору класса `Employee` (строка 9) для инициализации скрытого элемента, недоступного напрямую в производном классе.
- Функция `earnings` (строка 27) переопределяет чистую виртуальную функцию `earnings` класса `Employee`, предоставляя ее конкретную реализацию, которая возвращает значение зарплаты за неделю. Если бы эта функция не была переопределена, класс `SalariedEmployee` унаследовал бы чистую виртуальную функцию `earnings` класса `Employee` и был бы абстрактным.

- Функция `toString` (строки 30–33) переопределяет функцию `toString` класса `Employee`. Если бы эта функция не была переопределена, она была бы унаследована от класса `Employee` и возвращала бы строку, содержащую только имя сотрудника. Переопределенная функция `toString` класса `SalariedEmployee` возвращает результат вызова `Employee::toString()` и значение зарплаты.

В заголовке класса `SalariedEmployee` функции `earnings` и `toString` объявлены с ключевым словом `override`, что дает уверенность в их правильном переопределении. Напомним, что они являются виртуальными в базовом классе `Employee`, поэтому остаются виртуальными во всей иерархии классов.

```
 1 // Fig. 10.14: SalariedEmployee.cpp
 2 // Определения функций класса SalariedEmployee
 3 #include <fmt/format.h>
 4 #include <stdexcept>
 5 #include "SalariedEmployee.h" // Определение класса SalariedEmployee
 6
 7 // Конструктор
 8 SalariedEmployee::SalariedEmployee(std::string_view name, double salary)
 9    : Employee{name} {
10    setSalary(salary);
11 }
12
13 // Записываем оклад
14 void SalariedEmployee::setSalary(double salary) {
15    if (salary < 0.0) {
16       throw std::invalid_argument("Weekly salary must be >= 0.0");
17    }
18
19    m_salary = salary;
20 }
21
22 // Возвращаем оклад
23 double SalariedEmployee::getSalary() const {return m_salary;}
24
25 // Рассчитываем зарплату,
26 // переопределяя чистую виртуальную функцию earnings класса Employee
27 double SalariedEmployee::earnings() const {return getSalary();}
28
29 // Возвращаем строковое представление объекта SalariedEmployee
30 std::string SalariedEmployee::toString() const {
31    return fmt::format("{}\n{}: ${:.2f}", Employee::toString(),
32       "salary", getSalary());
33 }
```

Илл. 10.14 | Файл реализации класса SalariedEmployee

10.9.3. Создание конкретного производного класса CommissionEmployee

Класс `CommissionEmployee` (илл. 10.15–10.16) наследуется от класса `Employee` (илл. 10.15, строка 8). Файл реализации функций (илл. 10.16) содержит:

- конструктор (строки 8–12), который принимает имя сотрудника, объем продаж и комиссионную ставку, а затем передает имя конструктору класса `Employee` (строка 9) для инициализации унаследованных элементов данных;
- *set*-функции (строки 15–21 и 27–34) для записи новых значений в элементы данных `m_grossSales` и `m_commissionRate`;
- *get*-функции (строки 24 и 37–39) для возврата значений `m_grossSales` и `m_commissionRate`;
- переопределенную функцию `earnings` (строки 42–44), вычисляющую зарплату сотрудника;
- переопределенную функцию `toString` (строки 47–51), возвращающую результат вызова `Employee::toString()`, объем продаж и комиссионную ставку.

```
1 // Fig. 10.15: CommissionEmployee.h
2 // Класс CommissionEmployee, производный от Employee
3 #pragma once
4 #include <string>
5 #include <string_view>
6 #include "Employee.h" // Определение класса Employee
7
8 class CommissionEmployee final : public Employee {
9 public:
10    CommissionEmployee(std::string_view name, double grossSales,
11       double commissionRate);
12    virtual ~CommissionEmployee() = default; // Виртуальный деструктор
13
14    void setGrossSales(double grossSales);
15    double getGrossSales() const;
16
17    void setCommissionRate(double commissionRate);
18    double getCommissionRate() const;
19
20    // Ключевое слово override указывает на переопределение
21    double earnings() const override; // Расчет зарплаты
22    std::string toString() const override; // Строковое представление
23 private:
24    double m_grossSales{0.0};
25    double m_commissionRate{0.0};
26 };
```

Илл. 10.15 | Заголовок класса CommissionEmployee

```
// Fig. 10.16: CommissionEmployee.cpp
// Определения функций класса CommissionEmployee
#include <fmt/format.h>
#include <stdexcept>
#include "CommissionEmployee.h" // Определение класса CommissionEmployee

// Конструктор
CommissionEmployee::CommissionEmployee(std::string_view name,
   double grossSales, double commissionRate) : Employee(name) {
   setGrossSales(grossSales);
   setCommissionRate(commissionRate);
}

// Записываем объем продаж
void CommissionEmployee::setGrossSales(double grossSales) {
   if (grossSales < 0.0) {
      throw std::invalid_argument("Gross sales must be >= 0.0");
   }

   m_grossSales = grossSales;
}

// Возвращаем объем продаж
double CommissionEmployee::getGrossSales() const {return m_grossSales;}

// Записываем комиссионную ставку
void CommissionEmployee::setCommissionRate(double commissionRate) {
   if (commissionRate <= 0.0 || commissionRate >= 1.0) {
      throw std::invalid_argument(
         "Commission rate must be > 0.0 and < 1.0");
   }

   m_commissionRate = commissionRate;
}

// Возвращаем комиссионную ставку
double CommissionEmployee::getCommissionRate() const {
   return m_commissionRate;
}

// Рассчитываем зарплату
double CommissionEmployee::earnings() const {
   return getGrossSales() * getCommissionRate();
}

// Возвращаем строковое представление объекта CommissionEmployee
std::string CommissionEmployee::toString() const {
   return fmt::format("{}\n{}: ${:.2f}\n{}: {:.2f}", Employee::toString(),
      "gross sales", getGrossSales(),
      "commission rate", getCommissionRate());
}
```

Илл. 10.16 | Файл реализации класса CommissionEmployee

10.9.4. Демонстрация полиморфных операций с объектами

Программа, предназначенная для тестирования нашей иерархии (илл. 10.17), создает объекты конкретных классов **SalariedEmployee** и **CommissionEmployee** и сначала обращается к этим объектам по их именам, а затем методом динамического полиморфизма: через вектор указателей базового класса **Employee**. Объекты создаются в строках 16–17. В строках 20–23 выводится информация о каждом сотруднике и его зарплате. В этих строках обращение к объектам выполняется через имена переменных, поэтому компилятор может определить во время компиляции тип каждого объекта, чтобы выбрать, какие из функций **toString** и **earnings** должны быть вызваны.

```
 1 // fig10_17.cpp
 2 // Обработка объектов класса, производного от Employee, по именам переменных,
 3 // а затем полиморфно: через указатели и ссылки базового класса
 4 #include <fmt/format.h>
 5 #include <iostream>
 6 #include <vector>
 7 #include "Employee.h"
 8 #include "SalariedEmployee.h"
 9 #include "CommissionEmployee.h"
10
11 void virtualViaPointer(const Employee* baseClassPtr); // Прототип
12 void virtualViaReference(const Employee& baseClassRef); // Прототип
13
14 int main() {
15    // Создаем объекты производного класса
16    SalariedEmployee salaried{"John Smith", 800.0};
17    CommissionEmployee commission{"Sue Jones", 10000, .06};
18
19    // Выводим данные каждого объекта Employee
20    std::cout << "EMPLOYEES PROCESSED INDIVIDUALLY VIA VARIABLE NAMES\n"
21       << fmt::format("{}\n{}{:.2f}\n\n{}\n{}{:.2f}\n\n",
22             salaried.toString(), "earned $", salaried.earnings(),
23             commission.toString(), "earned $", commission.earnings());
24
25    // Создаем и инициализируем вектор указателей базового класса
26    std::vector<Employee*> employees{&salaried, &commission};
27
28    std::cout << "EMPLOYEES PROCESSED POLYMORPHICALLY VIA"
29       << " DYNAMIC BINDING\n\n";
30
31    // Вызываем virtualViaPointer для вывода данных каждого
32    // объекта Employee с помощью динамического связывания
33    std::cout << "VIRTUAL FUNCTION CALLS MADE VIA BASE-CLASS POINTERS\n";
34
35    for (const Employee* employeePtr : employees) {
36       virtualViaPointer(employeePtr);
37    }
38
39    // Вызываем virtualViaReference для вывода данных каждого
40    // объекта Employee с помощью динамического связывания
41    std::cout << "VIRTUAL FUNCTION CALLS MADE VIA BASE-CLASS REFERENCES\n";
```

Окончание ⇨

```
42
43     for (const Employee* employeePtr : employees) {
44        virtualViaReference(*employeePtr); // Разыменованный указатель!
45     }
46 }
47
48 // Вызываем виртуальные функции toString и earnings класса Employee
49 // через указатель базового класса с помощью динамического связывания
50 void virtualViaPointer(const Employee* baseClassPtr) {
51     std::cout << fmt::format("{}\nearned ${:.2f}\n\n",
52        baseClassPtr->toString(), baseClassPtr->earnings());
53 }
54
55 // Вызываем виртуальные функции toString и earnings класса Employee
56 // через ссылку базового класса с помощью динамического связывания
57 void virtualViaReference(const Employee& baseClassRef) {
58     std::cout << fmt::format("{}\nearned ${:.2f}\n\n",
59        baseClassRef.toString(), baseClassRef.earnings());
60 }
```

```
EMPLOYEES PROCESSED INDIVIDUALLY VIA VARIABLE NAMES
name: John Smith
salary: $800.00
earned $800.00

name: Sue Jones
gross sales: $10000.00
commission rate: 0.06
earned $600.00

EMPLOYEES PROCESSED POLYMORPHICALLY VIA DYNAMIC BINDING

VIRTUAL FUNCTION CALLS MADE VIA BASE-CLASS POINTERS
name: John Smith
salary: $800.00
earned $800.00

name: Sue Jones
gross sales: $10000.00
commission rate: 0.06
earned $600.00

VIRTUAL FUNCTION CALLS MADE VIA BASE-CLASS REFERENCES
name: John Smith
salary: $800.00
earned $800.00

name: Sue Jones
gross sales: $10000.00
commission rate: 0.06
earned $600.00
```

Илл. 10.17 | Обработка объектов класса, производного от Employee, через имена переменных, а затем полиморфно: через указатели и ссылки базового класса

Создание вектора указателей

В строке 26 создается и инициализируется вектор `employees`, содержащий два указателя типа `Employee`: на объекты `salaried` и `commission`. Компилятор позволяет инициализировать элементы вектора адресами этих объектов, поскольку `SalariedEmployee` является `Employee` и `CommissionEmployee` тоже является `Employee`.

Функция virtualViaPointer

В строках 35–37 перебираются элементы вектора `employees`, при этом функция `virtualViaPointer` (строки 50–53) поочередно вызывается с каждым элементом в качестве аргумента. Она получает в свой параметр `baseClassPtr` тот адрес, который хранится в выбранном элементе вектора `employees`, и вызывает через этот указатель виртуальные функции `toString` и `earnings`. У функции `virtualViaPointer` нет никакой информации о типе объекта (`SalariedEmployee` или `CommissionEmployee`), она знает только о базовом классе `Employee`. Программа по очереди направляет указатель `baseClassPtr` на разные конкретные объекты производных классов, поэтому компилятор не знает, какие функции конкретных классов должны быть вызваны через `baseClassPtr`. Компилятор должен разрешить выбор вызова во время выполнения программы с использованием динамического связывания. Во время выполнения программы каждый вызов виртуальной функции приводит к корректному вызову подходящей функции для того объекта, на который указывает `baseClassPtr`. Выходные данные программы показывают, что функции каждого класса вызываются правильно и выводят корректные данные. Получение данных о зарплате методом динамического полиморфизма дает тот же результат, что и прямой доступ к данным в строках 22 и 23.

Функция virtualViaReference

В строках 43–45 перебираются элементы вектора `employees`, при этом функция `virtualViaReference` (строки 57–60) поочередно вызывается с каждым его элементом в качестве аргумента. Она получает в свой параметр `baseClassRef` (типа `const Employee&`) ссылку на объект, которая образуется путем разыменования указателя, хранящегося в выбранном элементе вектора `employees` (строка 44). При каждом вызове этой функции вызываются виртуальные функции `toString` и `earnings` через `baseClassRef`. Вы можете убедиться, что полиморфная обработка данных работает и со ссылками базового класса. Каждый вызов виртуальной функции приводит к корректному вызову подходящей функции для того объекта, на который ссылается `baseClassRef`. Это еще один пример динамического связывания.

Итак, через ссылки базового класса мы получили такие же выходные данные, как двумя предыдущими способами: через указатели базового класса и через имена переменных (со статическим связыванием).

10.10. Динамический полиморфизм, виртуальные функции и динамическое связывание «под капотом» компилятора

Давайте «поднимем капот» компилятора C++ и посмотрим на внутреннюю реализацию динамического полиморфизма, виртуальных функций и динамического связывания. Мы рассмотрим один из возможных вариантов реализации, чтобы вы лучше поняли, как все это работает. Важнее всего, чтобы вы учитывали повышенную нагрузку на компьютер. Динамический полиморфизм требует дополнительных ресурсов: памяти и процессорного времени. Поэтому в одних случаях он вам поможет, а в других случаях его надо избегать. Например, классы стандартной библиотеки C++ должны работать очень быстро, поэтому они, как правило, реализованы без виртуальных функций.

Сначала мы расскажем о структурах данных, которые компилятор создает для поддержки динамического полиморфизма. Вы увидите, что для этого нужна тройная косвенность, то есть три уровня указателей. Затем мы покажем, как эти структуры данных используются во время выполнения программы для работы виртуальных функций и для динамического связывания.

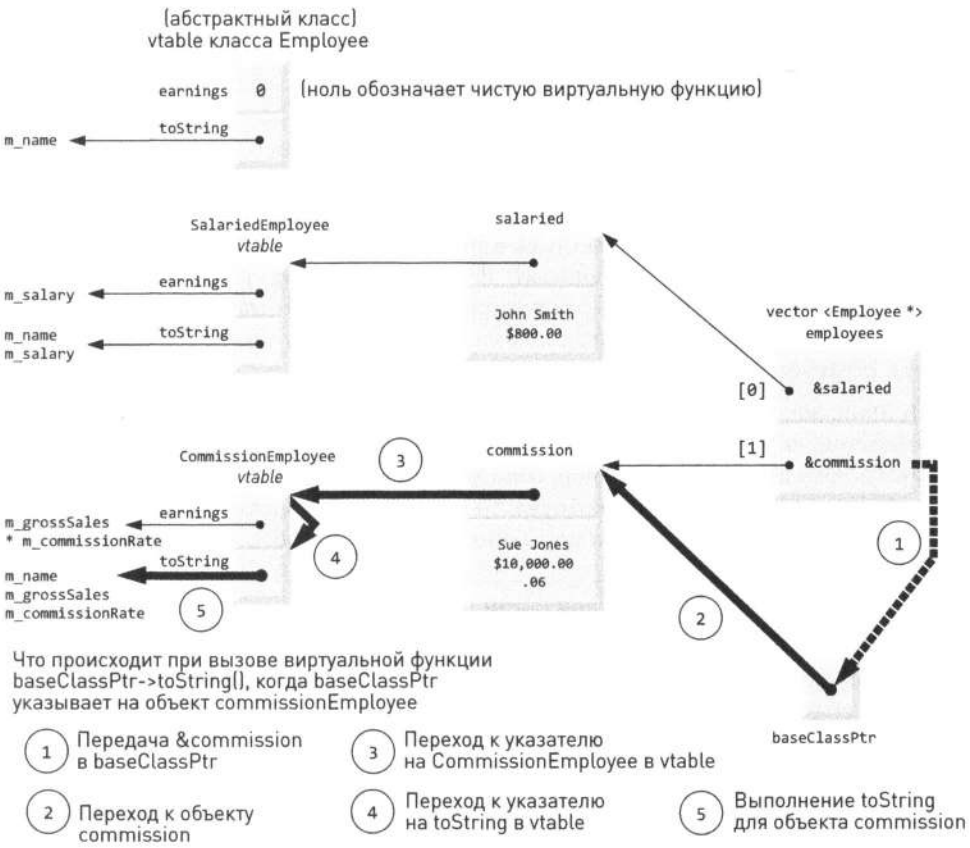
Таблицы виртуальных функций

Когда в C++ компилируется класс с одной или несколькими виртуальными функциями, для него создается **таблица виртуальных функций (virtual function table**, или сокращенно **vtable**). Она содержит указатели на виртуальные функции класса. Каждый такой указатель хранит адрес памяти, с которого начинается машинный код, выполняющий задачу функции. Вы уже знаете, что имя массива неявно преобразуется в адрес первого элемента массива. Так и имя функции неявно преобразуется в начальный адрес кода функции.

Для динамического связывания выполняемая программа обращается к виртуальной таблице класса, чтобы при каждом вызове виртуальной функции выбирать реализацию, подходящую для объекта соответствующего класса. В левом столбце на илл. 10.18 показаны виртуальные таблицы наших классов `Employee`, `SalariedEmployee` и `CommissionEmployee`.

Таблица виртуальных функций класса Employee

Первый указатель на функцию в таблице виртуальных функций класса `Employee` содержит `0`, поскольку `earnings` — это чистая виртуальная функция без реализации. Второй указатель указывает на функцию `toString`, возвращающую строку с именем сотрудника. На этой иллюстрации мы для экономии места сократили выходные данные каждой функции `toString`. Любой класс, в таблице виртуальных функций которого есть одна или несколько чистых виртуальных функций (представленных значением `0`), является абстрактным классом. Для сравнения, `SalariedEmployee` и `CommissionEmployee` — это конкретные классы, поэтому в их таблицах виртуальных функций нет `nullptr`.



Илл. 10.18 | Как устроены вызовы виртуальных функций

Таблица виртуальных функций класса SalariedEmployee

Указатель на функцию `earnings` в виртуальной таблице класса `SalariedEmployee` указывает на переопределенную этим классом функцию `earnings`, возвращающую зарплату сотрудника. Класс `SalariedEmployee` также переопределяет функцию `toString`, поэтому соответствующий указатель ссылается на функцию `toString` из класса `SalariedEmployee`, возвращающую имя и зарплату сотрудника.

Таблица виртуальных функций класса CommissionEmployee

Указатель на функцию `earnings` в виртуальной таблице класса `CommissionEmployee` указывает на функцию `earnings` из класса `CommissionEmployee`, возвращающую произведение объема продаж сотрудника и комиссионной ставки. Указатель функции `toString` ссылается на версию этой функции из класса `CommissionEmployee`, возвращающую имя сотрудника, комиссионную ставку и объем продаж. Как и в слу-

чае с классом `SalariedEmployee`, обе функции переопределяют функции базового класса `Employee`.

Наследование конкретных виртуальных функций

Каждый конкретный класс в нашей программе предоставляет реализацию виртуальной функции `earnings` и функции `toString`. Учитывая, что `earnings` — это чистая виртуальная функция, все прямые производные классы от класса `Employee` должны ее реализовать, чтобы стать конкретными. Но прямым производным классам не обязательно реализовывать функцию `toString`, чтобы стать конкретными, — они могут наследовать реализацию функции `toString` от класса `Employee`. В нашем случае оба производных класса переопределяют функцию `toString` из класса `Employee`.

Если производный класс в иерархии наследует функцию, не переопределяя ее, то в таблице виртуальных функций этого класса хранится указатель на унаследованную реализацию. Например, если мы решим не переопределять функцию `toString` в классе `CommissionEmployee`, то в таблице этого класса будет указатель на ту же версию `toString`, что и в таблице класса `Employee`.

Три уровня указателей для реализации полиморфизма

Динамический полиморфизм может быть реализован с помощью элегантной структуры данных, включающей три уровня указателей. Мы рассмотрели первый уровень: указатели в таблицах виртуальных функций. Каждый такой указатель содержит адрес кода, который должен выполняться при вызове соответствующей виртуальной функции.

В среднем столбце диаграммы показан второй уровень указателей. Создавая объект с одной или несколькими виртуальными функциями, компилятор «прикрепляет» к нему указатель на таблицу виртуальных функций класса. Обычно этот указатель находится в начале объекта, но возможны и другие варианты в зависимости от компилятора. На диаграмме вы видите эти указатели в объектах классов `SalariedEmployee` и `CommissionEmployee` (илл. 10.17). Рядом показаны значения элементов данных каждого объекта.

Третий уровень указателей содержит адреса объектов, для которых будут вызываться виртуальные функции. В правом столбце диаграммы показан вектор `employees`, содержащий эти указатели.

Теперь рассмотрим, как выполняется типичный вызов виртуальной функции. Предположим, что из функции `virtualViaPointer` выполняется вызов `baseClassPtr->toString()` (илл. 10.17, строка 52). Предположим, что `baseClassPtr` в данный момент указывает на `employees[1]`, то есть на адрес объекта `commission` в векторе. При компиляции этой инструкции компилятор видит, что вызов выполняется через указатель базового класса, а функция `toString` является виртуальной. Поэтому компилятор находит `toString` во второй строке таблицы виртуальных функций и берет из этой строки адрес машинного кода функции.

Сгенерированный компилятором код выполнит следующие операции (их порядковые номера указаны в кружках на илл. 10.18):

1. Возьмет из вектора `employees` элемент с индексом `1` (адрес объекта `commission`) и передаст его в качестве аргумента функции `virtualViaPointer`. Ее параметр `baseClassPtr` станет указателем на объект `commission`.
2. Разыменует этот указатель для перехода к объекту `commission`, который начинается с указателя на таблицу виртуальных функций класса `CommissionEmployee`.
3. Разыменует этот указатель для перехода к таблице виртуальных функций класса `CommissionEmployee`.
4. Получит указатель на функцию `toString` из второй строки таблицы виртуальных функций класса `CommissionEmployee`[1].
5. Выполнит функцию `toString` для объекта `commission`, которая возвратит строку с именем сотрудника, объемом продаж и комиссионной ставкой.

Структуры данных на диаграмме с таблицами виртуальных функций могут показаться сложными. Но компилятор выполняет все эти сложные операции, скрывая их от вас, поэтому с точки зрения программиста применять динамический полиморфизм очень просто. Но не забывайте, что при каждом вызове виртуальной функции выполняются операции разыменования указателей и обращения к памяти, требующие дополнительного процессорного времени. А таблицы виртуальных функций и добавляемые к объектам указатели на них требуют дополнительной памяти.

Динамический полиморфизм, обычно реализуемый в C++ с помощью виртуальных функций и динамического связывания, очень эффективен. Вы можете использовать его в большинстве программ, не опасаясь существенной нагрузки на компьютер. И лишь в некоторых случаях эта дополнительная нагрузка будет играть решающую роль. Она может оказаться чрезмерной в программах реального времени, которые должны работать максимально быстро, а также в программах с огромным количеством маленьких объектов, по сравнению с которыми дополнительные указатели были бы слишком большими.

10.11. Идиома невиртуального интерфейса

Идиома невиртуального интерфейса (non-virtual interface idiom, или сокращенно **NVI)**[2,3] — это еще один способ реализовать динамический полиморфизм на основе иерархий классов. Херб Саттер (Herb Sutter) впервые предложил NVI в своей

[1] На практике операции 3 и 4 могут быть объединены в одной машинной инструкции.

[2] «Non-Virtual Interface (NVI) Pattern». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Non-virtual_interface_pattern.

[3] Мариус Банчила (Marius Bancila), Modern C++ Programming Cookbook: Master C++ Core Language and Standard Library Features, with over 100 Recipes, Updated to C++20. Бирмингем: издательство Packt, 2020, с. 562–567.

статье «Virtuality»[1]. В этом разделе мы будем следовать четырем рекомендациям Саттера по реализации иерархий классов:

1. «Делайте интерфейсы невиртуальными, используя метод шаблонов» — это паттерн объектно-ориентированного проектирования. Саттер объясняет, что открытая (`public`) виртуальная функция служит двум целям: описывает часть интерфейса класса и позволяет переопределять себя в производных классах для настройки поведения. Он рекомендует, чтобы каждая функция служила только какой-либо одной из этих целей.
2. «Делайте виртуальные функции скрытыми (`private`)». Производный класс может переопределять скрытые виртуальные функции своего базового класса. Сокрытие виртуальной функции гарантирует, что она служит только одной цели: позволяет переопределять себя в производных классах для настройки поведения. Невиртуальная функция в базовом классе внутренне вызывает скрытую виртуальную функцию как деталь реализации.
3. «Только если производным классам надо вызывать базовую реализацию виртуальной функции, делайте эту виртуальную функцию защищенной (`protected`)». Это позволяет производным классам не только переопределять виртуальную функцию базового класса, но и пользоваться ее реализацией в базовом классе, чтобы избежать повторения кода в производных классах.
4. «Деструктор базового класса должен быть либо открытым и виртуальным, либо защищенным и невиртуальным». Здесь мы снова сделаем деструктор базового класса `Employee` открытым и виртуальным. В главе 11 вы увидите, что открытые виртуальные деструкторы важны при удалении динамически выделяемых объектов производного класса через указатели базового класса.

Статья Саттера демонстрирует каждую рекомендацию, подчеркивая необходимость отделения интерфейса класса от его реализации, что является хорошей практикой программирования.

Рефакторинг класса Employee с идиомой невиртуального интерфейса

Давайте доработаем класс `Employee` (илл. 10.19–10.20), основываясь на рекомендациях Саттера. Большая часть кода останется такой же, как в разделе 10.9, поэтому мы сосредоточимся на описании изменений. Перечислим основные изменения в заголовке (илл. 10.19):

- Открытые функции `earnings` (строка 15) и `toString` (строка 16) больше не являются виртуальными, а функция `earnings`, как вы скоро увидите, будет реализована в классе `Employee`. Теперь каждая из функций `earnings` и `toString` служит только одной цели: позволяет клиентскому коду получать данные из объектов класса `Employee` (размер зарплаты и строковое представление). В прежних версиях эти функции служили еще и в качестве

[1] Херб Саттер (Herb Sutter), «Virtuality», C/C++ Users Journal, том 19, № 9, сентябрь 2001 г. Был доступен: 15 сентября 2023 г. http://www.gotw.ca/publications/mill18.htm.

точек настройки для производных классов, но теперь для этого предназначены новые функции.

- Мы добавили защищенную (`protected`) виртуальную функцию `getString` (строка 18), которая будет точкой настройки для производных классов. Эта функция будет вызываться невиртуальной функцией `toString`. Защищенные элементы базового класса доступны производным классам. Как вы скоро увидите, функции `getString` производных классов будут и вызывать, и переопределять функцию `getString` класса `Employee`.
- Мы добавили скрытую (`private`) чистую виртуальную функцию `getPay` (строка 21), которая будет точкой настройки для производных классов. Эта функция будет вызываться невиртуальной функцией `earnings`. Производные классы будут переопределять функцию `getPay` для настройки расчетов зарплаты.

```
1 // Fig. 10.19: Employee.h
2 // Абстрактный базовый класс Employee
3 #pragma once // Предотвращаем повторные включения заголовка
4 #include <string>
5 #include <string_view>
6
7 class Employee {
8 public:
9    Employee(std::string_view name);
10   virtual ~Employee() = default;
11
12   void setName(std::string_view name);
13   std::string getName() const;
14
15   double earnings() const; // Невиртуальная функция
16   std::string toString() const; // Невиртуальная функция
17 protected:
18   virtual std::string getString() const; // Виртуальная функция
19 private:
20   std::string m_name;
21   virtual double getPay() const = 0; // Чистая виртуальная функция
22 };
```

Илл. 10.19 | Абстрактный базовый класс Employee

Перечислим основные изменения в файле реализации функций класса `Employee` (илл. 10.20):

- Функция `earnings` (строка 17) теперь предоставляет конкретную реализацию, возвращающую результат вызова скрытой чистой виртуальной функции `getPay`. Этот вызов разрешен, потому что во время выполнения программы функция `earnings` будет вызываться для объекта конкретного класса, производного от `Employee`.

- Функция `toString` (строка 20) теперь возвращает результат вызова защищенной виртуальной функции `getString`.
- Мы добавили определение новой защищенной функции `getString`, возвращающей строковое представление объекта по умолчанию, содержащее имя сотрудника. Эта функция является защищенной, поэтому производные классы могут переопределять ее, а также вызывать ее, чтобы получить ту часть строкового представления, которая относится к базовому классу.

```
1 // Fig. 10.20: Employee.cpp
2 // Определения функций абстрактного базового класса Employee
3 // Внимание: для чистых виртуальных функций нет определений
4 #include <fmt/format.h>
5 #include "Employee.h" // Определение класса Employee
6
7 // Конструктор
8 Employee::Employee(std::string_view name) : m_name{name} {} // Пустое тело
9
10 // Записываем имя сотрудника
11 void Employee::setName(std::string_view name) {m_name = name;}
12
13 // Получаем имя сотрудника
14 std::string Employee::getName() const {return m_name;}
15
16 // Открытая невиртуальная функция, возвращающая зарплату сотрудника
17 double Employee::earnings() const {return getPay();}
18
19 // Открытая невиртуальная функция, возвращающая строковое представление
      объекта Employee
20 std::string Employee::toString() const {return getString();}
21
22 // Защищенная виртуальная функция, которую производные классы могут
      переопределять и вызывать
23 std::string Employee::getString() const {
24    return fmt::format("name: {}", getName());
25 }
```

Илл. 10.20 | Определения функций абстрактного базового класса Employee

Доработанный класс SalariedEmployee

Перечислим основные изменения в классе `SalariedEmployee` (илл. 10.21–10.22):

- В заголовке этого класса (илл. 10.21) больше нет прототипов функций `earnings` и `toString`. Эти открытые невиртуальные функции базового класса теперь наследуются от класса `Employee`.
- В секции `private` теперь объявлены переопределения скрытой чистой виртуальной функции базового класса `getPay` и защищенной виртуальной функции `getString` (илл. 10.21, строки 19–20). Функция `getString` клас-

са `SalariedEmployee` является скрытой, потому что это финальный (`final`) класс, который не может стать базовым.

- В файле реализации функций (илл. 10.22) теперь есть переопределенная функция `getPay` (строка 27), возвращающая размер зарплаты, и переопределенная функция `getString` (строки 30–33), возвращающая строковое представление объекта `SalariedEmployee`. Обратите внимание, что функция `getString` класса `SalariedEmployee` вызывает защищенную функцию `getString` класса `Employee`, чтобы получить соответствующую часть строкового представления.

```
1 // Fig. 10.21: SalariedEmployee.h
2 // Класс SalariedEmployee, производный от Employee
3 #pragma once
4 #include <string> // Класс string стандартной библиотеки C++
5 #include <string_view>
6 #include "Employee.h" // Определение класса Employee
7
8 class SalariedEmployee final : public Employee {
9 public:
10    SalariedEmployee(std::string_view name, double salary);
11    virtual ~SalariedEmployee() = default; // Виртуальный деструктор
12
13    void setSalary(double salary);
14    double getSalary() const;
15 private:
16    double m_salary{0.0};
17
18    // Ключевое слово override указывает на переопределение
19    double getPay() const override; // Расчет зарплаты
20    std::string getString() const override; // Строковое представление
21 };
```

Илл. 10.21 | Класс SalariedEmployee, производный от класса Employee

```
1 // Fig. 10.22: SalariedEmployee.cpp
2 // Определения функций класса SalariedEmployee
3 #include <fmt/format.h>
4 #include <stdexcept>
5 #include "SalariedEmployee.h" // Определение класса SalariedEmployee
6
7 // Конструктор
8 SalariedEmployee::SalariedEmployee(std::string_view name, double salary)
9    : Employee{name} {
10   setSalary(salary);
11 }
12
13 // Записываем оклад
14 void SalariedEmployee::setSalary(double salary) {
15    if (salary < 0.0) {
16       throw std::invalid_argument("Weekly salary must be >= 0.0");
```

Окончание ⇨

```
17    }
18
19    m_salary = salary;
20 }
21
22 // Возвращаем оклад
23 double SalariedEmployee::getSalary() const {return m_salary;}
24
25 // Рассчитываем зарплату;
26 // переопределяем чистую виртуальную функцию getPay класса Employee
27 double SalariedEmployee::getPay() const {return getSalary();}
28
29 // Возвращаем строковое представление объекта SalariedEmployee
30 std::string SalariedEmployee::getString() const {
31    return fmt::format("{}\n{}: ${:.2f}", Employee::getString(),
32       "salary", getSalary());
33 }
```

Илл. 10.22 | Определения функций класса SalariedEmployee

Доработанный класс CommissionEmployee

Перечислим основные изменения в классе `CommissionEmployee` (илл. 10.23–10.24):

- В заголовке этого класса (илл. 10.23) больше нет прототипов функций `earnings` и `toString`. Эти открытые невиртуальные функции базового класса теперь наследуются от класса `Employee`.
- В секции `private` теперь объявлены переопределения скрытой чистой виртуальной функции базового класса `getPay` и защищенной виртуальной функции `getString` (илл. 10.23, строки 24–25). Опять же, функция `getString` является скрытой, потому что это финальный (`final`) класс, который не может стать базовым.
- В файле реализации функций (илл. 10.24) теперь есть переопределенная функция `getPay` (строки 42–44), возвращающая результат расчета комиссионного вознаграждения, и переопределенная функция `getString` (строки 47–51), возвращающая строковое представление объекта `CommissionEmployee`. Функция `getString` вызывает защищенную функцию `getString` класса `Employee`, чтобы получить соответствующую часть строкового представления.

```
1 // Fig. 10.23: CommissionEmployee.h
2 // Класс CommissionEmployee, производный от класса Employee
3 #pragma once
4 #include <string>
5 #include <string_view>
6 #include "Employee.h" // Определение класса Employee
7
8 class CommissionEmployee final : public Employee {
9 public:
10    CommissionEmployee(std::string_view name, double grossSales,
```

```
11       double commissionRate);
12    virtual ~CommissionEmployee() = default; // Виртуальный деструктор
13
14    void setGrossSales(double grossSales);
15    double getGrossSales() const;
16
17    void setCommissionRate(double commissionRate);
18    double getCommissionRate() const;
19 private:
20    double m_grossSales{0.0};
21    double m_commissionRate{0.0};
22
23    // Ключевое слово override указывает на переопределение
24    double getPay() const override; // Расчет зарплаты
25    std::string getString() const override; // Строковое представление
26 };
```

Илл. 10.23 | Класс CommissionEmployee, производный от класса Employee

```
 1 // Fig. 10.24: CommissionEmployee.cpp
 2 // Определения функций класса CommissionEmployee
 3 #include <fmt/format.h>
 4 #include <stdexcept>
 5 #include "CommissionEmployee.h" // Определение класса CommissionEmployee
 6
 7 // Конструктор
 8 CommissionEmployee::CommissionEmployee(std::string_view name,
 9    double grossSales, double commissionRate) : Employee{name} {
10    setGrossSales(grossSales);
11    setCommissionRate(commissionRate);
12 }
13
14 // Записываем объем продаж
15 void CommissionEmployee::setGrossSales(double grossSales) {
16    if (grossSales < 0.0) {
17       throw std::invalid_argument("Gross sales must be >= 0.0");
18    }
19
20    m_grossSales = grossSales;
21 }
22
23 // Возвращаем объем продаж
24 double CommissionEmployee::getGrossSales() const {return m_grossSales;}
25
26 // Записываем комиссионную ставку
27 void CommissionEmployee::setCommissionRate(double commissionRate) {
28    if (commissionRate <= 0.0 || commissionRate >= 1.0) {
29       throw std::invalid_argument(
30          "Commission rate must be > 0.0 and < 1.0");
31    }
32
```

Окончание ⇨

```
33    m_commissionRate = commissionRate;
34 }
35
36 // Возвращаем комиссионную ставку
37 double CommissionEmployee::getCommissionRate() const {
38    return m_commissionRate;
39 }
40
41 // Рассчитываем зарплату
42 double CommissionEmployee::getPay() const {
43    return getGrossSales() * getCommissionRate();
44 }
45
46 // Возвращаем строковое представление объекта CommissionEmployee
47 std::string CommissionEmployee::getString() const {
48    return fmt::format(
49       "{}\n{}: ${:.2f}\n{}: {:.2f}", Employee::getString(),
50       "gross sales", getGrossSales(),
51       "commission rate", getCommissionRate());
52 }
```

Илл. 10.24 | Определения функций класса CommissionEmployee

Динамический полиморфизм в иерархии Employee с идиомой невиртуального интерфейса

Тестовая программа для этого примера идентична предыдущей (илл. 10.17), поэтому на илл. 10.25 мы показываем только выходные данные. Как видите, они тоже идентичны выходным данным предыдущей программы (илл. 10.17). Это подтверждает, что код остается полиморфным даже с защищенными и скрытыми виртуальными функциями базового класса. Наш клиентский код теперь вызывает только невиртуальные функции. Тем не менее каждый производный класс обеспечивает требуемое поведение, переопределяя защищенные и скрытые виртуальные функции базового класса. Теперь виртуальные функции являются деталями реализации внутри иерархии классов, скрытыми от программиста клиентского кода. Мы можем изменять эти реализации виртуальных функций (и, возможно, даже их сигнатуры) без изменения клиентского кода!

```
EMPLOYEES PROCESSED INDIVIDUALLY VIA VARIABLE NAMES
name: John Smith
salary: $800.00
earned $800.00

name: Sue Jones
gross sales: $10000.00
commission rate: 0.06
earned $600.00

EMPLOYEES PROCESSED POLYMORPHICALLY VIA DYNAMIC BINDING
```

```
VIRTUAL FUNCTION CALLS MADE VIA BASE-CLASS POINTERS
name: John Smith
salary: $800.00
earned $800.00

name: Sue Jones
gross sales: $10000.00
commission rate: 0.06
earned $600.00

VIRTUAL FUNCTION CALLS MADE VIA BASE-CLASS REFERENCES
name: John Smith
salary: $800.00
earned $800.00

name: Sue Jones
gross sales: $10000.00
commission rate: 0.06
earned $600.00
```

Илл. 10.25 | Обработка объектов класса, производного от Employee, со статическим связыванием, а затем полиморфно: с динамическим связыванием

10.12. Наследование интерфейса вместо наследования реализации[1]

Наследование реализации в основном используется для определения родственных классов со множеством одинаковых элементов данных и функций. Такое наследование создает тесно связанные классы, в которых элементы данных и функции базового класса наследуются в производных классах. Изменения в базовом классе напрямую влияют на все соответствующие производные классы.

Тесные связи между классами могут затруднять модификацию иерархии. Подумайте, как бы вы могли дополнить иерархию класса `Employee` из раздела 10.9 для поддержки пенсионных накоплений. В США есть разные планы пенсионных накоплений, например «401(k)» или «IRA». Вы могли бы добавить в класс `Employee` чистую виртуальную функцию `makeRetirementDeposit` и определить соответствующие производные классы: `SalariedEmployeeWith401k`, `SalariedEmployeeWithIRA`, `CommissionEmployeeWith401k`, `CommissionEmployeeWithIRA` и т. д., в каждом из которых будет собственная реализация чистой виртуальной функции `makeRetirementDeposit`. Очевидно, что при таком подходе вы получите много производных классов, поэтому реализация и обслуживание этой иерархии окажется непростой задачей.

[1] Эрих Гамма и др., «Приемы объектно-ориентированного проектирования». Также обсуждается: Джошуа Блох (Joshua Bloch), «Effective Java». Upper Saddle River, Нью-Джерси: Эддисон-Уэсли, 2008.

Маленькие иерархии, находящиеся под контролем одного программиста, как правило, легче модифицировать, чем большие иерархии, поддерживаемые командами программистов. Это верно даже для иерархий с тесно связанными классами, которые образуются при наследовании реализации.

Переосмысление иерархии Employee: композиция и внедрение зависимостей

Год за годом программисты пишут многочисленные статьи и посты в блогах о проблемах тесных связей в иерархиях классов (как в иерархии `Employee` из раздела 10.9). В следующем примере мы выполним рефакторинг нашей иерархии `Employee`. Теперь модель расчета зарплаты не будет жестко встроена в иерархию[1].

Чтобы добиться этого, мы используем композицию и внедрение зависимостей: добавим в класс указатель на объект, от которого зависит требуемое поведение объектов этого класса. В нашем примере поведение — это способ расчета зарплаты каждого сотрудника. Мы определим новый класс `Employee`, содержащий указатель на объект класса `CompensationModel` с функциями `earnings` и `toString`. У этого класса `Employee` не будет производных классов: мы объявим его финальным (`final`). Затем мы определим производные от `CompensationModel` классы, реализующие две модели расчета зарплаты сотрудников:

- фиксированная зарплата (оклад);
- комиссионное вознаграждение, зависящее от объема продаж.

Конечно, мы можем аналогично определить и любые другие модели расчета зарплаты.

Наследование интерфейса дает самые гибкие возможности

Для создания моделей расчета зарплаты мы будем использовать **наследование интерфейса (interface inheritance)**. Каждый конкретный класс `CompensationModel` будет производным от класса, содержащего только чистые виртуальные функции. Класс, в котором все функции являются чистыми виртуальными, называется **чистым абстрактным классом (pure abstract class)**, или **интерфейсом (interface)**. В C++ Core Guidelines рекомендуется отдавать предпочтение наследованию от чистых абстрактных классов, а не от классов с деталями реализации[2,3,4,5].

---

[1] Мы благодарим Брайана Гетца (Brian Goetz), архитектора языка Java, который предложил рассматриваемую в этом разделе архитектуру, когда рецензировал нашу книгу «Java How to Program».

[2] «I.25: Prefer Abstract Classes as Interfaces to Class Hierarchies». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-abstract.

[3] «C.121: If a Base Class Is Used as an Interface, Make It a Pure Abstract Class». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-abstract.

[4] «C.122: Use Abstract Classes As Interfaces When Complete Separation of Interface and Implementation Is Needed». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-separation.

[5] «C.129: When Designing a Class Hierarchy, Distinguish Between Implementation Inheritance and Interface Inheritance». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-kind.

В интерфейсах обычно нет никаких элементов данных. В отличие от наследования реализации, наследование интерфейса может потребовать от вас больше работы, поскольку конкретные классы должны предоставлять и элементы данных, и реализации чистых виртуальных функций, даже если все они похожи или идентичны во многих конкретных классах. Зато, как мы покажем в конце раздела, этот подход дает вам больше гибкости благодаря отсутствию тесных связей между классами.

Интерфейсы упрощают модификацию сложного программного обеспечения. В разделе 10.8.2 мы объяснили это на хорошем примере с драйверами устройств для операционных систем.

10.12.1. Переосмысление иерархии класса Employee: интерфейс CompensationModel

Рассмотрим иерархию класса `Employee` из раздела 10.9 с композицией и интерфейсом. Можно сказать, что каждый объект `Employee` содержит `CompensationModel`. На илл. 10.26 вы видите определение интерфейса `CompensationModel`, который является чистым абстрактным классом, поэтому не имеет файла `.cpp`. В этом классе есть сгенерированный компилятором виртуальный деструктор и две чистые виртуальные функции:

- `earnings` — для расчета зарплаты сотрудника на основе его `CompensationModel`;
- `toString` — для создания строкового представления `CompensationModel`.

Любой класс, унаследованный от `CompensationModel` и переопределяющий его чистые виртуальные функции, является классом `CompensationModel`, реализующим этот интерфейс.

```
1 // Fig. 10.26: CompensationModel.h
2 // Интерфейс (чистый абстрактный класс) CompensationModel
3 #pragma once // Предотвращаем повторные включения заголовка
4 #include <string>
5
6 class CompensationModel {
7 public:
8    virtual ~CompensationModel() = default; // Сгенерированный деструктор
9    virtual double earnings() const = 0; // Чистая виртуальная функция
10   virtual std::string toString() const = 0; // Чистая виртуальная функция
11 };
```

Илл. 10.26 | Интерфейс (чистый абстрактный класс) CompensationModel

10.12.2. Класс Employee

На илл. 10.27 показан наш новый класс `Employee`. Каждый объект `Employee` содержит указатель на соответствующую реализацию `CompensationModel` (строка 16). Обратите внимание, что этот класс объявлен финальным (`final`), поэтому его нельзя использовать в качестве базового.

```
1 // Fig. 10.27: Employee.h
2 // Класс Employee содержит CompensationModel
3 #pragma once // Предотвращаем повторные включения заголовка
4 #include <string>
5 #include <string_view>
6 #include "CompensationModel.h"
7
8 class Employee final {
9 public:
10    Employee(std::string_view name, CompensationModel* modelPtr);
11    void setCompensationModel(CompensationModel *modelPtr);
12    double earnings() const;
13    std::string toString() const;
14 private:
15    std::string m_name{};
16    CompensationModel* m_modelPtr{}; // Указатель на объект реализации
17 };
```

Илл. 10.27 | Класс Employee содержит CompensationModel

В коде, показанном на илл. 10.28, определены функции класса `Employee`:

- Конструктор (строки 10–11) инициализирует имя сотрудника и направляет свой указатель `CompensationModel` на объект, реализующий интерфейс `CompensationModel`. Этот метод известен как **внедрение конструктора (constructor injection)**. Конструктор получает указатель (или ссылку) на другой объект и сохраняет его в создаваемом объекте[1].
- Функция `setCompensationModel` (строки 15–17) позволяет клиентскому коду изменять `CompensationModel` сотрудника, направляя указатель `m_modelPtr` на другой объект `CompensationModel`. Этот метод известен как **внедрение свойств (property injection)**.
- Функция `earnings` (строки 20–22) определяет зарплату сотрудника, вызывая из соответствующей реализации `CompensationModel` функцию `earnings` через указатель `m_modelPtr`.
- Функция `toString` (строки 25–27) создает строковое представление объекта `Employee` (с именем сотрудника и сведениями о зарплате), вызывая из соответствующей реализации `CompensationModel` функцию `toString` через указатель `m_modelPtr`.

Внедрение конструктора и внедрение свойств — это две формы **внедрения зависимостей (dependency injection)**. Вы определяете часть поведения объекта, предоставляя указатель или ссылку на другой объект, от которого зависит это

[1] При внедрении зависимостей объект CompensationModel должен существовать дольше объекта Employee, чтобы предотвратить логическую ошибку во время выполнения, известную как висячий указатель. Мы обсуждаем висячие указатели в разделе 11.6.

поведение[1]. В нашем примере поведение, позволяющее рассчитывать зарплату и генерировать строковое представление, зависит от `CompensationModel`.

```
1 // Fig. 10.28: Employee.cpp
2 // Определения функций класса Employee
3 #include <fmt/format.h>
4 #include <string>
5 #include "CompensationModel.h"
6 #include "Employee.h"
7
8 // Внедряем конструктор для инициализации
9 // указателя CompensationModel на реализацию CompensationModel
10 Employee::Employee(std::string_view name, CompensationModel* modelPtr)
11    : m_name{name}, m_modelPtr{modelPtr} {}
12
13 // Set-функция выполняет внедрение свойств для направления
14 // указателя CompensationModel на новую реализацию CompensationModel
15 void Employee::setCompensationModel(CompensationModel* modelPtr) {
16    m_modelPtr = modelPtr;
17 }
18
19 // Используем CompensationModel для расчета зарплаты
20 double Employee::earnings() const {
21    return m_modelPtr->earnings();
22 };
23
24 // Возвращаем строковое представление объекта Employee
25 std::string Employee::toString() const {
26    return fmt::format("{}\n{}", m_name, m_modelPtr->toString());
27 }
```

Илл. 10.28 | Определения функций класса Employee

10.12.3. Реализации CompensationModel

Теперь мы определим модели расчета зарплаты для разных категорий сотрудников, то есть реализации классов, производных от класса `CompensationModel`. Объекты этих классов будут внедрены в объекты `Employee` для расчета зарплаты каждой категории сотрудников.

Salaried — класс, производный от CompensationModel

Модель расчета зарплаты класса `Salaried` (илл. 10.29–10.30) предназначена для сотрудников, получающих фиксированную зарплату (оклад). Этот класс содержит элемент данных `m_salary` и переопределяет функции `earnings` и `toString` из интерфейса `CompensationModel`. В строке 7 класс `Salaried` объявлен финальным (`final`), поэтому он является последним звеном в иерархии `CompensationModel`, его нельзя использовать в качестве базового.

[1] «Dependency Injection». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Dependency_injection.

```
1 // Fig. 10.29: Salaried.h
2 // Класс Salaried реализует интерфейс CompensationModel
3 #pragma once
4 #include <string>
5 #include "CompensationModel.h" // Определение класса CompensationModel
6
7 class Salaried final : public CompensationModel {
8 public:
9    explicit Salaried(double salary);
10   double earnings() const override;
11   std::string toString() const override;
12 private:
13   double m_salary{0.0};
14 };
```

Илл. 10.29 | Класс Salaried реализует интерфейс CompensationModel

```
1 // Fig. 10.30: Salaried.cpp
2 // Определения функций класса Salaried
3 #include <fmt/format.h>
4 #include <stdexcept>
5 #include "Salaried.h" // Определение класса Salaried
6
7 // Конструктор
8 Salaried::Salaried(double salary) : m_salary{salary} {
9    if (m_salary < 0.0) {
10      throw std::invalid_argument("Weekly salary must be >= 0.0");
11 }
12 }
13
14 // Переопределяем чистую виртуальную функцию earnings
15 double Salaried::earnings() const {return m_salary;}
16
17 // Переопределяем чистую виртуальную функцию toString
18 std::string Salaried::toString() const {
19    return fmt::format("salary: ${:.2f}", m_salary);
20 }
```

Илл. 10.30 | Определения функций класса Salaried

Commission — класс, производный от CompensationModel

Модель оплаты труда класса `Commission` (илл. 10.31–10.32) предназначена для сотрудников отдела продаж, получающих в качестве зарплаты комиссионное вознаграждение, зависящее от объема продаж. Этот класс содержит элементы данных `m_grossSales` и `m_commissionRate` и переопределяет функции `earnings` и `toString` из интерфейса `CompensationModel`. Как и класс `Salaried`, класс `Commission` объявлен финальным (строка 7), поэтому является последним звеном в иерархии `CompensationModel`, его нельзя использовать в качестве базового.

```
// Fig. 10.31: Commission.h
// Класс Commission реализует интерфейс CompensationModel
#pragma once
#include <string>
#include "CompensationModel.h" // Определение класса CompensationModel

class Commission final : public CompensationModel {
public:
   Commission(double grossSales, double commissionRate);
   double earnings() const override;
   std::string toString() const override;
private:
   double m_grossSales{0.0};
   double m_commissionRate{0.0};
};
```

Илл. 10.31 | Класс Commission реализует интерфейс CompensationModel

```
// Fig. 10.32: Commission.cpp
// Определения функций класса Commission
#include <fmt/format.h>
#include <stdexcept>
#include "Commission.h" // Определение класса Commission

// Конструктор
Commission::Commission(double grossSales, double commissionRate)
   : m_grossSales{grossSales}, m_commissionRate{commissionRate} {

if (m_grossSales < 0.0) {
   throw std::invalid_argument("Gross sales must be >= 0.0");
}

if (m_commissionRate <= 0.0 || m_commissionRate >= 1.0) {
   throw std::invalid_argument(
      "Commission rate must be > 0.0 and < 1.0");
}
}

// Переопределяем чистую виртуальную функцию earnings
double Commission::earnings() const {
   return m_grossSales * m_commissionRate;
}

// Переопределяем чистую виртуальную функцию toString
std::string Commission::toString() const {
   return fmt::format("gross sales: ${:.2f}; commission rate: {:.2f}",
      m_grossSales, m_commissionRate);
}
```

Илл. 10.32 | Определения функций класса Commission

10.12.4. Тест-драйв новой иерархии

Итак, мы создали интерфейс `CompensationModel` и две реализации производных классов: две модели расчета зарплаты. Теперь мы создадим объекты класса `Employee` и инициализируем их конкретными реализациями `CompensationModel` (илл. 10.33).

```
 1 // fig10_33.cpp
 2 // Расчет зарплаты на основе реализаций CompensationModel
 3 #include <fmt/format.h>
 4 #include <iostream>
 5 #include <vector>
 6 #include "Employee.h"
 7 #include "Salaried.h"
 8 #include "Commission.h"
 9
10 int main() {
11    // Создаем объекты, представляющие сотрудников и модели расчета зарплаты
12    Salaried salaried{800.0};
13    Employee salariedEmployee{"John Smith", &salaried};
14
15    Commission commission{10000, .06};
16    Employee commissionEmployee{"Sue Jones", &commission};
17
18    // Создаем и инициализируем вектор объектов Employee
19    std::vector employees{salariedEmployee, commissionEmployee};
20
21    // Выводим на экран строковые представления объектов Employee
22    for (const Employee& employee : employees) {
23       std::cout << fmt::format("{}\nearned: ${:.2f}\n\n",
24          employee.toString(), employee.earnings());
25    }
26 }
```

```
John Smith
salary: $800.00
earned: $800.00

Sue Jones
gross sales: $10000.00; commission rate: 0.06
earned: $600.00
```

Илл. 10.33 | Расчет зарплаты на основе реализаций CompensationModel

В строке 12 создается объект класса `Salaried` (`salaried`). В строке 13 создается объект `salariedEmployee`, в который внедрена модель расчета фиксированной зарплаты: указатель на объект `salaried` является вторым аргументом конструктора. В строке 15 создается объект класса `Commission` (`commission`). В строке 16 создается объект `commissionEmployee`, в который внедрена модель расчета комиссионной зарплаты: указатель на объект `commission` является вторым аргументом конструктора[1].

[1] В этом примере объекты CompensationModel переживут свои соответствующие объекты Employee, поэтому исключен риск возникновения висячего указателя.

В строке 19 создается вектор объектов `Employee`, инициализированный объектами `salariedEmployee` и `commissionEmployee`. Наконец, в строках 22–25 перебираются элементы вектора и на экран выводится строковое представление каждого объекта `Employee` и зарплата обоих сотрудников.

10.12.5. Преимущества внедрения зависимостей

Простота модификации при изменениях моделей расчета зарплаты

Поскольку все модели расчета зарплаты — это отдельные классы, реализующие один и тот же интерфейс `CompensationModel`, мы можем легко вносить в систему изменения по мере необходимости. Предположим, что в компании появились новые модели оплаты труда. Для каждого из них достаточно определить новый производный от `CompensationModel` класс с соответствующей функцией `earnings`.

Простота модификации при переходе сотрудников между отделами

Архитектура программы, основанная на интерфейсах, композиции и внедрении зависимостей, является более гибкой, чем прежняя, описанная в разделе 10.9. Там в случае изменения категории сотрудника вам пришлось бы изменить тип объекта, представляющего этого сотрудника. Для этого надо было бы создать новый объект соответствующего класса, производного от `Employee`, и переместить данные о сотруднике в новый объект. А благодаря внедрению зависимостей достаточно вызвать функцию `setCompensationModel` и перенаправить указатель с одной реализации `CompensationModel` на другую.

Простота модификации при расширении класса Employee

Архитектура программы, основанная на интерфейсах, композиции и внедрении зависимостей, упрощает расширение класса `Employee`. Например, мы решим добавить поддержку пенсионных планов «401(k)» и «IRA». В этом случае мы можем сказать, что каждый объект `Employee` содержит («has-a») `RetirementPlan`. Сначала мы определим интерфейс `RetirementPlan` с функцией `makeRetirementDeposit`, а также его реализации в производных классах. После этого надо будет внести в класс `Employee` лишь небольшие изменения:

- добавить элемент данных, указывающий на `RetirementPlan`;
- добавить аргумент конструктора для инициализации указателя на `RetirementPlan`;
- добавить функцию `setRetirementPlan`, позволяющую изменять пенсионный план сотрудника.

10.13. Полиморфизм на основе шаблона класса std::variant и функции std::visit

Итак, мы реализовали динамический полиморфизм методами наследования реализации и наследования интерфейса. Оба метода требуют наличия иерархии

классов. А можно ли реализовать динамический полиморфизм применительно к объектам классов, не связанных между собой? Да, для этого в стандартной библиотеке (C++17) есть шаблон класса `std::variant` и функция `std::visit` из заголовка `<variant>`[1, 2, 3, 4]. Особенность этого метода заключается в том, что вы должны заранее знать все типы, которые понадобятся вашей программе для полиморфной обработки, — их называют **замкнутым набором типов (closed set of types)**. Объект `std::variant` может хранить только один объект любого типа, указанного при создании объекта `std::variant`. Как вы скоро увидите, доступ к функциям объектов, хранящихся в объектах `std::variant`, осуществляется через функцию `std::visit`.

Для демонстрации полиморфизма на основе шаблона класса `std::variant` мы доработаем пример из раздела 10.12. Классы останутся почти такими же, как прежде, поэтому мы укажем только на отличия.

Класс Salaried

Класс `Salaried` (илл. 10.34–10.35) определяет модель расчета фиксированной зарплаты для объектов класса `Employee`. Единственное отличие этого класса от предыдущего (описанного в разделе 10.12) заключается в том, что теперь класс `Salaried` не является производным. Таким образом, его функции `earnings` и `toString` не переопределяют виртуальные функции базового класса.

```
// Fig. 10.34: Salaried.h
// Класс Salaried
#pragma once
#include <string>

class Salaried {
public:
   Salaried(double salary);
   double earnings() const;
   std::string toString() const;
private:
   double m_salary{0.0};
};
```

Илл. 10.34 | Класс Salaried

[1] Невин Либер (Nevin Liber), «The Many Variants of std::variant». Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=JUxhwf7gYLg.

[2] «std::variant». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/utility/variant.

[3] Бартломей Филипек (Bartlomiej Filipek), «Runtime Polymorphism with std::variant and std::visit». Был доступен: 15 сентября 2023 г. https://www.bfilipek.com/2020/04/variant-virtualpolymorphism.html.

[4] Бартломей Филипек, «Everything You Need to Know About std::variant from C++17». Был доступен: 15 сентября 2023 г. https://www.bfilipek.com/2018/06/variant.html.

```
// Fig. 10.35: Salaried.cpp
// Определения функций класса Salaried
#include <fmt/format.h>
#include <stdexcept>
#include "Salaried.h" // Определение класса Salaried

// Конструктор
Salaried::Salaried(double salary) : m_salary{salary} {
   if (m_salary < 0.0) {
      throw std::invalid_argument("Weekly salary must be >= 0.0");
   }
}

// Рассчитываем зарплату
double Salaried::earnings() const {return m_salary;}

// Возвращаем строковое представление объекта
std::string Salaried::toString() const {
   return fmt::format("salary: ${:.2f}", m_salary);
}
```

Илл. 10.35 | Определения функций класса Salaried

Класс Commission

Класс `Commission` (илл. 10.36–10.37) определяет модель расчета зарплаты для тех объектов класса `Employee`, для которых рассчитывается комиссионное вознаграждение в зависимости от объема продаж. Как и класс `Salaried`, класс `Commission` не является производным. Таким образом, его функции `earnings` и `toString` не переопределяют виртуальные функции базового класса, как в разделе 10.12.

```
// Fig. 10.36: Commission.h
// Класс Commission
#pragma once
#include <string>

class Commission {
public:
   Commission(double grossSales, double commissionRate);
   double earnings() const;
   std::string toString() const;
private:
   double m_grossSales{0.0};
   double m_commissionRate{0.0};
};
```

Илл. 10.36 | Класс Commission

```
1 // Fig. 10.37: Commission.cpp
2 // Определения функций класса Commission
3 #include <fmt/format.h>
4 #include <stdexcept>
5 #include "Commission.h" // Определение класса Commission
6
7 // Конструктор
8 Commission::Commission(double grossSales, double commissionRate)
9    : m_grossSales{grossSales}, m_commissionRate{commissionRate} {
10
11    if (m_grossSales < 0.0) {
12       throw std::invalid_argument("Gross sales must be >= 0.0");
13    }
14
15    if (m_commissionRate <= 0.0 || m_commissionRate >= 1.0) {
16       throw std::invalid_argument(
17          "Commission rate must be > 0.0 and < 1.0");
18    }
19 }
20
21 // Рассчитываем зарплату
22    double Commission::earnings() const {
23       return m_grossSales * m_commissionRate;
24 }
25
26 // Возвращаем строковое представление объекта
27 std::string Commission::toString() const {
28    return fmt::format("gross sales: ${:.2f}; commission rate: {:.2f}",
29       m_grossSales, m_commissionRate);
30 }
```

Илл. 10.37 | Определения функций класса Commission

Определение класса Employee

Как и в разделе 10.12, каждый объект `Employee` (илл. 10.38) содержит модель расчета зарплаты (строка 21). Однако в этом примере модель расчета представляет собой объект `std::variant`, содержащий либо объект класса `Commission`, либо объект класса `Salaried`, — обратите внимание, что это объект, а не указатель на объект. В строке 11 есть объявление `using`:

```
using CompensationModel = std::variant<Commission, Salaried>;
```

Оно объявляет псевдоним `CompensationModel` для типа `std::variant`:

```
std::variant<Commission, Salaried>
```

Такие объявления `using` (C++11) называются **объявлениями псевдонимов (alias declarations)**. Они позволяют создавать удобные имена для сложных типов, таких как `std::variant`, которые потенциально могут иметь много параметров. В любой момент времени объект нашего типа `std::variant` может хранить либо объект `Commission`, либо объект `Salaried`. В строке 21 мы используем псевдоним

`CompensationModel`, определяющий объект `std::variant`, в котором хранится модель расчета зарплаты.

```
// Fig. 10.38: Employee.h
// Класс Employee содержит CompensationModel
#pragma once // Предотвращаем повторные включения заголовка
#include <string>
#include <string_view>
#include <variant>
#include "Commission.h"
#include "Salaried.h"

// Определяем удобное имя для типа std::variant
using CompensationModel = std::variant<Commission, Salaried>;

class Employee {
public:
   Employee(std::string_view name, CompensationModel model);
   void setCompensationModel(CompensationModel model);
   double earnings() const;
   std::string toString() const;
private:
   std::string m_name{};
   CompensationModel m_model; // Внимание: это не указатель!
};
```

Илл. 10.38 | Класс Employee содержит CompensationModel

Типобезопасное объединение

В языках C и C++ **объединение (union)**[1] — это объект, в котором могут поочередно храниться объекты разных типов. Все элементы объединения совместно используют одну и ту же область памяти, поэтому объединение может содержать в любой момент времени не более одного объекта и требует достаточного объема памяти для самого большого из своих возможных элементов. Объект типа `std::variant` часто называют **типобезопасным объединением (type-safe union)**[2].

Конструктор класса Employee и функция setCompensationModel

Функции класса `Employee` (илл. 10.39) выполняют те же задачи, что и аналогичные функции на илл. 10.28, но с несколькими изменениями. Конструктор класса `Employee` (строки 8–9) и функция `setCompensationModel` (строки 12–14) получают объект `CompensationModel`. В отличие от архитектуры с композицией и внедрением зависимостей, в объекте `std::variant` хранится не указатель на объект, а сам объект.

[1] «Union Declaration». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/language/union.

[2] «std::variant». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/utility/variant.

```
1 // Fig. 10.39: Employee.cpp
2 // Определения функций класса Employee
3 #include <fmt/format.h>
4 #include <string>
5 #include "Employee.h"
6
7 // Конструктор
8 Employee::Employee(std::string_view name, CompensationModel model)
9    : m_name{name}, m_model{model} {}
10
11 // Изменяем CompensationModel
12 void Employee::setCompensationModel(CompensationModel model) {
13    m_model = model;
14 }
15
16 // Возвращаем оклад
17 double Employee::earnings() const {
18    auto getEarnings{[](const auto& model){return model.earnings();}};
19    return std::visit(getEarnings, m_model);
20 }
21
22 // Возвращаем строковое представление объекта Employee
23 std::string Employee::toString() const {
24    auto getString{[](const auto& model){return model.toString();}};
25    return fmt::format("{}\n{}", m_name, std::visit(getString, m_model));
26 }
```

Илл. 10.39 | Определения функций класса Employee

Вызов функций earnings и toString класса Employee через функцию std::visit

Ключевое различие между полиморфизмом на основе иерархии классов и полиморфизмом на основе `std::variant` заключается в том, что объект `std::variant` не может вызывать функции хранимого объекта. Для вызова этих функций вам надо использовать функцию `std::visit` стандартной библиотеки. Рассмотрим строки 18–19 в функции `earnings`:

```
auto getEarnings{[](const auto& model){return model.earnings();}};
return std::visit(getEarnings, m_model);
```

В строке 18 определяется переменная `getEarnings` и инициализируется обобщенным лямбда-выражением, которое получает ссылку на объект (`model`) и вызывает его функцию `earnings`. Это лямбда-выражение может вызывать функцию `earnings` любого объекта, которая не принимает аргументы и возвращает значение. В строке 19 лямбда-выражение `getEarnings` и объект `m_model` передаются в функцию `std::visit`. Функция `std::visit` передает `m_model` в лямбда-выражение и возвращает результат вызова функции `earnings` объекта `m_model`.

Аналогично в функции `toString` класса `Employee` (строка 24) создается лямбда-выражение, возвращающее результат вызова функции `toString` некоторого объекта.

В строке 25 вызывается функция `std::visit`, которая передает `m_model` в лямбда-выражение и возвращает результат вызова функции `toString` объекта `m_model`.

Тест-драйв динамического полиморфизма на основе std::variant и std::visit

В функции `main`, в строках 11–12 (илл. 10.40), создаются два объекта класса `Employee`. Первый хранит в своем `std::variant` объект класса `Salaried`, а второй — объект класса `Commission`. Выражение в строке 11

```
Salaried{800.0}
```

создает объект `Salaried`, который немедленно используется для инициализации `CompansationModel` объекта `Employee`. Аналогично выражение в строке 12

```
Commission{10000.0, .06}
```

создает объект `Commission`, который немедленно используется для инициализации `CompansationModel` объекта `Employee`.

```
 1 // fig10_40.cpp
 2 // Расчет зарплаты для разных категорий сотрудников
 3 #include <fmt/format.h>
 4 #include <iostream>
 5 #include <vector>
 6 #include "Employee.h"
 7 #include "Salaried.h"
 8 #include "Commission.h"
 9
10 int main() {
11    Employee salariedEmployee{"John Smith", Salaried{800.0}};
12    Employee commissionEmployee{"Sue Jones", Commission{10000.0, .06}};
13
14    // Создаем и инициализируем вектор объектов Employee
15    std::vector employees{salariedEmployee, commissionEmployee};
16
17    // Выводим на экран данные о сотрудниках и их зарплате
18    for (const Employee& employee : employees) {
19       std::cout << fmt::format("{}\nearned: ${:.2f}\n\n",
20       employee.toString(), employee.earnings());
21    }
22 }
```

```
John Smith
salary: $800.00
earned: $800.00

Sue Jones
gross sales: $10000.00; commission rate: 0.06
earned: $600.00
```

Илл. 10.40 | Расчет зарплаты для разных категорий сотрудников

В строке 15 создается вектор объектов `Employee`. В строках 18–21 перебираются элементы вектора, при этом вызываются функции `earnings` и `toString` каждого объекта. Мы реализовали полиморфную обработку и получили такие же результаты, как в разделе 10.12. Возможность вызывать функциональность, общую для многих объектов, не связанных иерархией классов, часто называют **утиной типизацией (duck typing)**:

Если это ходит как утка и крякает как утка, то это должно быть уткой[1].

Иными словами, если объект имеет соответствующие функции и его тип является элементом `std::variant`, то объект будет работать в этом коде.

10.14. Множественное наследование

Мы рассмотрели одиночное наследование, при котором класс является производным только от одного базового класса. Но C++ поддерживает и множественное наследование: класс может наследовать элементы данных от двух и более базовых классов. Множественное наследование — это сложный инструмент, предназначенный только для опытных программистов. Некоторые проблемы множественного наследования настолько сложны, что новые языки программирования, такие как Java и C#, поддерживают только одиночное наследование[2]. Нужно быть крайне осторожным при проектировании системы с множественным наследованием. Его не следует использовать, если можно обойтись одиночным наследованием и (или) композицией.

При множественном наследовании в нескольких базовых классах могут оказаться элементы данных и функции с одинаковыми именами. Это создает проблему неоднозначной идентификации при компиляции. На сайте **isocpp.org**, в разделе часто задаваемых вопросов, рекомендуется выполнять множественное наследование только от чистых абстрактных базовых классов, чтобы избежать этой проблемы и других проблем, которые мы обсудим в этом разделе, а затем в разделе 10.14.1[3].

Пример множественного наследования

Рассмотрим пример множественного наследования реализации (илл. 10.41–10.45). Класс `Base1` (илл. 10.41) содержит:

- скрытый элемент данных типа `int` (`m_value`; строка 11);
- конструктор (строка 8), устанавливающий значение `m_value`;
- открытую функцию `getData` (строка 9), возвращающую значение `m_value`.

---

[1] «Утиная типизация». Был доступен: 15 сентября 2023 г. https://ru.wikipedia.org/wiki/Утиная_типизация.

[2] Точнее, Java и C# поддерживают только одиночное наследование реализации, допуская множественное наследование интерфейса.

[3] «Inheritance — Multiple and Virtual Inheritance». Был доступен: 15 сентября 2023 г. https://isocpp.org/wiki/faq/multiple-inheritance.

```
1 // Fig. 10.41: Base1.h
2 // Пример множественного наследования, Base1.h
3 #pragma once
4
5 // Определение класса Base1
6 class Base1 {
7 public:
8    explicit Base1(int value) : m_value{value} {}
9    int getData() const {return m_value;}
10 private: // Доступ для производных классов - через функцию getData
11    int m_value;
12 };
```

Илл. 10.41 | Пример множественного наследования, Base1.h

Класс `Base2` (илл. 10.42) похож на класс `Base1`, но его скрытый элемент данных имеет тип `char` (`m_letter`, строка 11). Подобно классу `Base1`, класс `Base2` содержит открытую функцию `getData`, но она возвращает значение переменной `m_letter`.

```
1 // Fig. 10.42: Base2.h
2 // Пример множественного наследования, Base2.h
3 #pragma once
4
5 // Определение класса Base2
6 class Base2 {
7 public:
8    explicit Base2(char letter) : m_letter{letter} {}
9    char getData() const {return m_letter;}
10 private: // Доступ для производных классов - через функцию getData
11    char m_letter;
12 };
```

Илл. 10.42 | Пример множественного наследования, Base2.h

Класс `Derived` (илл. 10.43–10.44) наследуется от классов `Base1` и `Base2`. Этот производный класс содержит:

- скрытый элемент данных `m_real` типа `double` (илл. 10.43, строка 18);
- конструктор для инициализации всех данных класса `Derived`;
- открытую функцию `getReal`, возвращающую значение `m_real`;
- открытую функцию `toString`, возвращающую строковое представление объекта `Derived`.

```
1 // Fig. 10.43: Derived.h
2 // Определение класса Derived, унаследованного от
3 // двух базовых классов: Base1 и Base2
4 #pragma once
5 #include <iostream>
6 #include <string>
```

Окончание ⇨

```
 7 #include "Base1.h"
 8 #include "Base2.h"
 9 using namespace std;
10
11 // Определение класса Derived
12 class Derived : public Base1, public Base2 {
13 public:
14    Derived(int value, char letter, double real);
15    double getReal() const;
16    std::string toString() const;
17 private:
18    double m_real; // Скрытый элемент данных производного класса
19 };
```

Илл. 10.43 | Пример множественного наследования, Derived.h

```
 1 // Fig. 10.44: Derived.cpp
 2 // Определения функций класса Derived
 3 #include <fmt/format.h> // Или в C++20: #include <format>
 4 #include "Derived.h"
 5
 6 // Конструктор для вызова конструкторов классов Base1 и Base2
 7 Derived::Derived(int value, char letter, double real)
 8    : Base1{value}, Base2{letter}, m_real{real} {}
 9
10 // Возвращаем значение m_real
11 double Derived::getReal() const {return m_real;}
12
13 // Выводим на экран элементы данных Derived
14 std::string Derived::toString() const {
15    return fmt::format("int: {}; char: {}; double: {}",
16       Base1::getData(), Base2::getData(), getReal());
17 }
```

Илл. 10.44 | Пример множественного наследования, Derived.cpp

Для множественного наследования (илл. 10.43) мы добавляем после имени производного класса и двоеточия (`Derived :`) список базовых классов, разделенный запятыми (строка 12). Обратите внимание, что на илл. 10.44 конструктор производного класса явно вызывает конструкторы каждого базового класса (`Base1` и `Base2`), используя синтаксис инициализации элементов (строка 8). Конструкторы базового класса вызываются в порядке наследования. Если список инициализаторов явно не вызывает конструктор базового класса, будет неявно вызван конструктор базового класса по умолчанию.

Решение проблемы одноименных функций базовых классов при множественном наследовании

Функция `toString` (строки 14–17) возвращает строковое представление объекта производного класса `Derived`, используя все *get*-функции производного класса.

Здесь могла бы возникнуть проблема неоднозначности, ведь объект `Derived` содержит две функции `getData`: одна из них унаследована от класса `Base1`, другая — от класса `Base2`. Но мы легко решили эту проблему с помощью оператора разрешения области видимости. Вызов функции `Base1::getData()` возвращает значение переменной, унаследованной от класса `Base1` (`m_value`), а вызов `Base2::getData()` возвращает значение переменной, унаследованной от класса `Base2` (`m_letter`).

Тест-драйв иерархии с множественным наследованием

На илл. 10.45 показана программа для тестирования нашей иерархии классов (илл. 10.41–10.44). В строке 10 создается объект `base1` класса `Base1` и инициализируется целым числом `10`. В строке 11 создается объект `base2` класса `Base2` и инициализируется символом `'Z'`. В строке 12 создается объект `derived` класса `Derived` и инициализируется значениями `7`, `'A'` и `3.5` соответственно.

```
1 // fig10_45.cpp
2 // Пример множественного наследования, тест-драйв
3 #include <fmt/format.h>
4 #include <iostream>
5 #include "Base1.h"
6 #include "Base2.h"
7 #include "Derived.h"
8
9 int main() {
10    Base1 base1{10}; // Объект базового класса Base1
11    Base2 base2{'Z'}; // Объект базового класса Base2
12    Derived derived{7, 'A', 3.5}; // Объект производного класса Derived
13
14    // Выводим на экран элементы данных всех объектов
15    std::cout << fmt::format("{}: {}\n{}: {}\n{}: {}\n\n",
16       "Object base1 contains", base1.getData(),
17       "Object base2 contains the character", base2.getData(),
18       "Object derived contains", derived.toString());
19
20    // Выводим на экран элементы данных объекта производного класса
21    // с помощью оператора разрешения области видимости
22    std::cout << fmt::format("{}\n{}: {}\n{}: {}\n{}: {}\n\n",
23       "Data members of Derived can be accessed individually:",
24       "int", derived.Base1::getData(),
25       "char", derived.Base2::getData(),
26       "double", derived.getReal());
27
28    std::cout << "Derived can be treated as an object"
29       << " of either base class:\n";
30
31    // Обрабатываем объект Derived как объект Base1
32    Base1* base1Ptr = &derived;
33    std::cout << fmt::format("base1Ptr->getData() yields {}\n",
34       base1Ptr->getData());
35
36    // Обрабатываем объект Derived как объект Base2
```

Окончание ⇨

```
37    Base2* base2Ptr = &derived;
38    std::cout << fmt::format("base2Ptr->getData() yields {}\n",
39       base2Ptr->getData());
40 }
```

```
Object base1 contains: 10
Object base2 contains the character: Z
Object derived contains: int: 7; char: A; double: 3.5

Data members of Derived can be accessed individually:
int: 7
char: A
double: 3.5

Derived can be treated as an object of either base class:
base1Ptr->getData() yields 7
base2Ptr->getData() yields A
```

Илл. 10.45 | Пример множественного наследования, тест-драйв

В строках 15–18 на экран выводятся данные каждого объекта. Для каждого из объектов `base1` и `base2` вызывается его функция `getData`. Несмотря на одинаковые имена функций, эти вызовы не являются неоднозначными. В строке 16 компилятор знает, что `base1` является объектом класса `Base1`, поэтому вызывается функция `getData` класса `Base1`. В строке 17 компилятор знает, что `base2` является объектом класса `Base2`, поэтому вызывается функция `getData` класса `Base2`. В строке 18 вызывается функция `toString` объекта `derived` для вывода на экран его данных.

В строках 22–26 данные объекта `derived` выводятся снова, но уже с помощью *get*-функций класса `Derived`. Здесь тоже могла бы возникнуть проблема неоднозначности, поскольку этот объект содержит две функции с именем `getData`: из класса `Base1` и из класса `Base2`. Выражение:

```
derived.Base1::getData()
```

получает значение `m_value`, унаследованное от класса `Base1`, а выражение:

```
derived.Base2::getData()
```

получает значение `m_letter`, унаследованное от класса `Base2`.

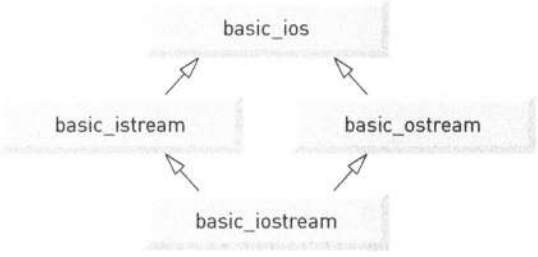
Отношение «является» при множественном наследовании

Отношение «является» («is-a»), свойственное одиночному наследованию, применимо и к множественному наследованию. В строке 32 мы присваиваем адрес объекта `derived` указателю `base1Ptr` типа `Base1`. Это разрешено, потому что объект `Derived` является объектом `Base1`. В строке 34 мы вызываем функцию `getData` класса `Base1` через указатель `base1Ptr`, чтобы получить значение элемента данных, унаследованного производным классом от класса `Base1`. В строке 37 мы присваиваем адрес объекта `derived` указателю `base2Ptr` типа `Base2`. Это разрешено, потому

что объект Derived является объектом Base2. В строке 39 мы вызываем функцию getData класса Base2 через указатель base2Ptr, чтобы получить значение элемента данных, унаследованного производным классом от класса Base2.

10.14.1. Ромбовидное наследование

В предыдущем примере мы рассмотрели множественное наследование, при котором один класс может быть производным от двух и более классов. Например, множественное наследование, показанное на следующей диаграмме, используется в стандартной библиотеке C++ для формирования класса basic_iostream:



Класс basic_ios является базовым для классов basic_istream и basic_ostream, которые формируются путем одиночного наследования. Класс basic_iostream является производным от basic_istream и от basic_ostream. Это позволяет объектам класса basic_iostream обеспечивать функциональность классов basic_istream и basic_ostream. Показанное на этой диаграмме наследование в иерархиях с множественным наследованием называется **ромбовидным наследованием (diamond inheritance)**.

Классы basic_istream и basic_ostream являются производными от класса basic_ios, поэтому у класса basic_iostream есть потенциальная проблема. Он может наследовать от basic_ios две копии элементов данных: одну — через класс basic_istream, а другую - через класс basic_ostream. Эта неоднозначность может вызвать ошибку компиляции, если компилятор не будет знать, какую копию элементов данных класса basic_ios надо использовать. Давайте рассмотрим, как эту проблему решают виртуальные базовые классы.

Ошибки компиляции, возникающие при неоднозначности в ромбовидном наследовании

На илл. 10.46 приведен пример неоднозначности, которая может возникнуть при ромбовидном наследовании. Класс Base (строки 7–10) содержит чистую виртуальную функцию print (строка 9). Классы DerivedOne (строки 13–17) и DerivedTwo (строки 20–24) открыто наследуют и переопределяют функцию print класса Base. Каждый из классов DerivedOne и DerivedTwo содержит **подобъект базового класса (base-class subobject)** — элемент класса Base.

```
// fig10_46.cpp
// Попытка полиморфного вызова функции, унаследованной
// от каждого из двух базовых классов
#include <iostream>

// Определение класса Base
class Base {
public:
   virtual void print() const = 0; // Чистая виртуальная функция
};

// Определение класса DerivedOne
class DerivedOne : public Base {
public:
   // Переопределяем функцию print
   void print() const override {std::cout << "DerivedOne\n";}
};

// Определение класса DerivedTwo
class DerivedTwo : public Base {
public:
   // Переопределяем функцию print
   void print() const override {std::cout << "DerivedTwo\n";}
};

// Определение класса Multiple
class Multiple : public DerivedOne, public DerivedTwo {
public:
   // Квалифицируем версию функции print
   void print() const override {DerivedTwo::print();}
};

int main() {
   Multiple both{}; // Создаем объект класса Multiple
   DerivedOne one{}; // Создаем объект класса DerivedOne
   DerivedTwo two{}; // Создаем объект класса DerivedTwo
   Base* array[3]{}; // Создаем массив указателей базового класса

   array[0] = &both; // Ошибка неоднозначности!
   array[1] = &one;
   array[2] = &two;

   // Полиморфно вызываем функцию print
   for (int i{0}; i < 3; ++i) {
      array[i] ->print();
   }
}
```

Сообщение компилятора Visual C++ об ошибке:

```
fig10_46.cpp(39,20): error C2594: '=': ambiguous conversions from 'Multiple *'
to 'Base *'
```

Илл. 10.46 | Попытка полиморфного вызова функции, унаследованной от каждого из двух базовых классов

Класс `Multiple` (строки 27–31) является производным от классов `DerivedOne` и `DerivedTwo`. Функция `print` класса `Multiple` переопределена для вызова функции `print` класса `DerivedTwo` (строка 30). Обратите внимание, что мы должны были квалифицировать вызов функции `print`. В данном случае мы использовали имя класса `DerivedTwo`, чтобы указать, какую версию функции `print` следует вызывать.

В функции `main` (строки 33–47) объявляются объекты классов `Multiple` (строка 34), `DerivedOne` (строка 35) и `DerivedTwo` (строка 36). В строке 37 объявляется традиционный массив указателей базового класса. Каждому элементу массива присваивается адрес объекта (строки 39–41). Ошибка возникает, когда адрес объекта `both` (объекта класса `Multiple`) присваивается элементу `array[0]`. Объект `both` содержит два подобъекта класса `Base`. Компилятор не знает, на какой из подобъектов должен указывать элемент `array[0]`, поэтому генерирует ошибку «ambiguous conversion».

10.14.2. Устранение конфликта подобъектов с помощью виртуального наследования базового класса

Проблема конфликта подобъектов решается **виртуальным наследованием (virtual inheritance)**. Если базовый класс наследуется как виртуальный, в производном классе появится только один подобъект. На илл. 10.47 показана модифицированная программа с виртуальными базовыми классами.

```
1 // fig10_47.cpp
2 // Использование виртуальных базовых классов
3 #include <iostream>
4
5 // Определение класса Base
6 class Base {
7 public:
8    virtual void print() const = 0; // Чистая виртуальная функция
9 };
10
11 // Определение класса DerivedOne
12 class DerivedOne : virtual public Base {
13 public:
14    // Переопределяем функцию print
15    void print() const override {std::cout << "DerivedOne\n";}
16 };
17
18 // Определение класса DerivedTwo
19 class DerivedTwo : virtual public Base {
20 public:
21    // Переопределяем функцию print
22    void print() const override {std::cout << "DerivedTwo\n";}
23 };
24
25 // Определение класса Multiple
26 class Multiple : public DerivedOne, public DerivedTwo {
27 public:
```

Окончание ⇨

```
28    // Квалифицируем версию функции print
29    void print() const override {DerivedTwo::print();}
30 };
31
32 int main() {
33    Multiple both; // Создаем объект класса Multiple
34    DerivedOne one; // Создаем объект класса DerivedOne
35    DerivedTwo two; // Создаем объект класса DerivedTwo
36    Base* array[3];
37
38    array[0] = &both; // Теперь разрешено
39    array[1] = &one;
40    array[2] = &two;
41
42    // Полиморфно вызываем функцию print
43    for (int i = 0; i < 3; ++i) {
44       array[i]->print();
45    }
46 }
```

```
DerivedTwo
DerivedOne
DerivedTwo
```

Илл. 10.47 | Использование виртуальных базовых классов

Ключевое изменение заключается в том, что классы `DerivedOne` (строка 12) и `DerivedTwo` (строка 19) наследуются от класса `Base` с ключевым словом `virtual`. Поскольку они являются производными от `Base`, каждый из них содержит подобъекты класса `Base`. Преимущество виртуального наследования еще не очевидно, пока класс `Multiple` не становится производным от классов `DerivedOne` и `DerivedTwo` (строка 26). Поскольку каждый из базовых классов унаследован как виртуальный, компилятор гарантирует, что класс `Multiple` наследует только один подобъект `Base`. Это предотвращает ошибку неоднозначности, показанную на илл. 10.46. Теперь компилятор разрешает неявное преобразование указателя производного класса (`&both`) в указатель базового класса `array[0]` в функции `main` (строка 38). Инструкция `for` в строках 43–45 полиморфно вызывает функцию `print` для каждого объекта.

Конструкторы в иерархиях множественного наследования с виртуальными базовыми классами

Реализация иерархии с виртуальными базовыми классами упрощается, если для базовых классов используются конструкторы по умолчанию. В предыдущем примере (илл. 10.46–10.47) используется конструктор по умолчанию, созданный компилятором. Если виртуальный базовый класс предоставляет конструктор, требующий аргументов, реализации производных классов становятся сложнее. Конечный производный класс должен явно вызывать конструктор виртуального базового класса. По этой причине рассмотрите возможность использования

конструкторов по умолчанию для виртуальных базовых классов, чтобы производным классам не приходилось явно вызывать пользовательские конструкторы виртуальных базовых классов[1].

10.15. Защищенные элементы класса

В главе 9 были представлены спецификаторы доступа `public` и `private`. Открытые (`public`) элементы базового класса доступны в этом классе, а также в любом месте программы, где есть доступ к какому-либо объекту этого класса или одного из его производных классов. Скрытые (`private`) элементы базового класса доступны только в теле этого класса, а также в дружественных ему классах и функциях. Элементы, объявленные со спецификатором доступа `protected`, называются **защищенными (protected)**. Этот спецификатор дает промежуточный уровень доступа между `public` и `private`. Защищенные элементы доступны в базовом классе, в его элементах, в дружественных ему классах и функциях, а также в классах и функциях, дружественных любым его производным классам.

При открытом наследовании все открытые и защищенные элементы базового класса сохраняют свой первоначальный уровень доступа, когда становятся элементами производного класса:

- открытые элементы базового класса становятся открытыми элементами производного класса;
- защищенные элементы базового класса становятся защищенными элементами производного класса.

Скрытые элементы базового класса недоступны за его пределами. Производные классы могут получить доступ к скрытым элементам базового класса только через открытые или защищенные функции, унаследованные от базового класса. Функции производного класса могут ссылаться на унаследованные от базового класса открытые и защищенные элементы по их именам.

Проблемы защищенных элементов данных

Желательно избегать защищенных элементов данных, потому что они могут стать причиной серьезных проблем:

- Объекту производного класса не обязательно использовать *set*-функцию для доступа к защищенному элементу базового класса. Таким образом, появляется возможность присвоить элементу недопустимое значение, после чего объект перейдет в некорректное состояние. Например, если элемент данных `m_grossSales` класса `CommissionEmployee` будет иметь спецификатор доступа

[1] «Inheritance — Multiple and Virtual Inheritance — What special considerations do I need to know about when I use virtual inheritance?». Был доступен: 15 сентября 2023 г. https://isocpp.org/wiki/faq/multiple-inheritance#virtual-inheritance-abcs.

`protected` (и этот класс не будет объявлен финальным), объект производного класса сможет присвоить элементу данных `m_grossSales` отрицательное значение.

- Функции производного класса обычно реализуются с зависимостью от данных базового класса. Производные классы должны зависеть только от тех функций базового класса, которые не являются скрытыми. Если мы изменим имя защищенного элемента данных базового класса, нам придется изменить каждый производный класс, который напрямую ссылается на этот элемент. Такие программы называют **хрупкими (fragile)** или **ломкими (brittle)**. Даже маленькое изменение в базовом классе может «сломать» все производные классы — это так называемая проблема хрупкого базового класса[1]. Она является основной причиной, почему в C++ Core Guidelines рекомендуется избегать защищенных данных[2]. Вы должны иметь возможность вносить изменения в базовый класс без необходимости изменять его производные классы.

В большинстве случаев предпочтительно использовать скрытые элементы данных, чтобы стимулировать правильную разработку программного обеспечения. Если элементы данных базового класса объявлены скрытыми, вы можете изменять реализацию базового класса без необходимости изменять реализацию производных классов. Это упрощает поддержку, модификацию и отладку вашего кода.

Производный класс может изменять состояние скрытых элементов данных базового класса только через унаследованные функции базового класса, которые не являются скрытыми. Если производный класс может получить доступ к скрытым элементам данных базового класса, то классы, производные от этого производного класса, тоже могут получить доступ к этим элементам данных, поэтому теряются преимущества инкапсуляции.

Защищенные функции базового класса

Одно из применений защищенных элементов данных — определение функций базового класса, которые не должны быть доступны в клиентском коде, но должны быть доступны в производных классах. В этом случае можно переопределить в производном классе защищенную виртуальную функцию базового класса и вызывать ее исходную версию (версию базового класса) из реализации производного класса[3]. Мы показали эту возможность в примере из раздела 10.11.

---

[1] «Хрупкий базовый класс». Был доступен: 15 сентября 2023 г. https://ru.wikipedia.org/wiki/Хрупкий_базовый_класс.

[2] C++ Core Guidelines, «C.133: Avoid protected Data». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-protected.

[3] Херб Саттер, «Virtuality». Был доступен: 15 сентября 2023 г. http://www.gotw.ca/publications/mill18.htm.

10.16. Открытое, защищенное и скрытое наследование

Вы можете выбрать между открытым, защищенным и скрытым наследованием, но открытое наследование является самым распространенным, а защищенное встречается крайне редко. В таблице приведена информация о доступности элементов данных базового класса в производном классе для каждого типа наследования.

Спецификатор доступа к элементам базового класса	Открытое (public) наследование	Защищенное (protected) наследование	Скрытое (private) наследование
`public`	`public` в производном классе. Прямой доступ из функций класса, дружественных функций и сторонних функций.	`protected` в производном классе. Прямой доступ из функций класса и дружественных функций.	`private` в производном классе. Прямой доступ из функций класса и дружественных функций.
`protected`	`protected` в производном классе. Прямой доступ из функций класса и дружественных функций.	`protected` в производном классе. Прямой доступ из функций класса и дружественных функций.	`private` в производном классе. Прямой доступ из функций класса и дружественных функций.
`private`	Нет доступа в производном классе. Доступ из функций класса и дружественных функций через открытые или защищенные функции класса, унаследованные от базового класса.	Нет доступа в производном классе. Доступ из функций класса и дружественных функций через открытые или защищенные функции класса, унаследованные от базового класса.	Нет доступа в производном классе. Доступ из функций класса и дружественных функций через открытые или защищенные функции класса, унаследованные от базового класса.

При открытом (`public`) наследовании:

- открытые элементы базового класса становятся открытыми элементами производного класса;
- защищенные элементы базового класса становятся защищенными элементами производного класса.

Скрытые элементы базового класса никогда не доступны напрямую в производном классе, но доступ к ним возможен через унаследованные открытые и защищенные функции базового класса, предназначенные для доступа к скрытым элементам.

При защищенном (`protected`) наследовании открытые и защищенные элементы базового класса становятся защищенными элементами производного класса. При скрытом (`private`) наследовании открытые и защищенные элементы базового класса становятся скрытыми элементами производного класса.

Для производных классов, созданных путем скрытого (`private`) и защищенного (`protected`) наследования, не действует отношение «является» применительно к их базовым классам, поскольку открытые элементы базовых классов недоступны для клиентского кода производных классов.

Наследование по умолчанию в классах и структурах

Как классы, так и структуры (рассмотренные в разделе 9.21) могут определять новые типы данных. Однако между классом и структурой есть два ключевых различия. Первое различие заключается в том, что по умолчанию элементы класса являются скрытыми, а элементы структуры являются открытыми. Другое отличие заключается в спецификаторе наследования по умолчанию. Например, следующее определение класса использует по умолчанию скрытое наследование:

```
class Derived : Base {
   // ...
};
```

Для сравнения, следующее определение структуры использует по умолчанию открытое наследование:

```
struct Derived : Base {
   // ...
};
```

10.17. Другие методы динамического полиморфизма и статический полиморфизм

В этой главе мы сосредоточились на наследовании, а также на нескольких методах динамического полиморфизма. Теперь мы кратко рассмотрим другие методы динамического полиморфизма, а также основы статического полиморфизма (**static polymorphism**), также называемого **полиморфизмом времени компиляции (compile-time polymorphism)**[1, 2]. Эти идиомы и методы предназначены для

[1] «C++: Static Polymorphism». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/C%2B%2B#Static_polymorphism.

[2] Катерина Бондаренко (Kateryna Bondarenko), «Static Polymorphism in C++». Был доступен: 15 сентября 2023 г. https://medium.com/@kateolenya/static-polymorphism-in-c-9c1ae27a945b.

опытных разработчиков и достаточно сложны. Некоторые из них мы продолжим рассматривать и в последующих главах. Дополнительную информацию по темам этого раздела вы найдете в соответствующих сносках.

10.17.1. Другие методы динамического полиморфизма

Для реализации динамического полиморфизма мы сначала использовали наследование и виртуальные функции. Затем мы использовали шаблон класса `std::variant` и функцию `std::visit` для реализации динамического полиморфизма с набором заранее известных типов, но без наследования и виртуальных функций. В этом разделе мы рассмотрим другие методы динамического полиморфизма.

Идиома концепта времени выполнения

Идиома концепта времени выполнения (runtime concept idiom)[1, 2] была предложена Шоном Парентом[3]. Она отделяет логику динамической полиморфной обработки от обрабатываемых типов. Эти типы не обязательно должны входить в иерархию классов, поэтому они не должны переопределять виртуальные функции базового класса.

Продолжим тему видеоигры из раздела 10.5. Менеджер экрана многократно удаляет и перерисовывает на экране игровые объекты. С точки зрения идиомы концепта времени выполнения «концепт»[4] — это функция `draw`, которая должна быть реализована для каждого типа изображаемых объектов. Но вместо того чтобы применять это требование с помощью базового класса, содержащего виртуальную функцию `draw`, идиома концепта времени выполнения использует несвязанные классы и утиную типизацию. Мы можем определить отдельные классы `Martian`, `Venusian`, `Plutonian`, `SpaceShip` и `LaserBeam`. В идиоме концепта времени выполнения все функции `draw` этих классов будут реализованы как перегруженные глобальные функции. У каждой функции будет аргумент: ссылка на тип объекта, который надо нарисовать. Такой подход устраняет тесные связи между классами в иерархиях наследования, затрудняющие внесение изменений в эти иерархии.

Чтобы реализовать полиморфное поведение, идиома концепта времени выполнения использует продуманную систему шаблонов классов, наследования и скрытых виртуальных функций для вызова автономных перегруженных функций `draw`. При этом шаблоны, наследование и виртуальные функции являются деталями реализации, скрытыми от программиста клиентского кода.

---

[1] Шон Парент (Sean Parent), «Inheritance Is the Base Class of Evil». Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=bIhUE5uUFOA.

[2] Шон Парент, «Better Code: Runtime Polymorphism». Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=QGcVXgEVMJgNDC.

[3] «Sean Parent». Был доступен: 15 сентября 2023 г. https://sean-parent.stlab.cc.

[4] Не следует путать с концептами C++20, о которых вы узнаете в главе 15.

Динамический полиморфизм со стиранием типов

В языке C++ **стирание типов (type erasure)**[1] — это основанный на шаблонах метод реализации полиморфизма с помощью **утиной типизации (duck typing)**. Шаблоны позволяют вам удалять (стирать) определенные типы из кода, который вы пишете. В отличие от метода на основе `std::variant` (раздел 10.13), метод стирания типов не требует заранее знать все типы, которые должны обрабатываться полиморфно. Идиома концепта времени выполнения использует «под капотом» стирание типов.

Метод стирания типов стал популярным, поэтому были разработаны библиотеки C++ с открытым исходным кодом, упрощающие написание программ с утиной типизацией. В частности, заслуживают внимания:

- библиотека `Folly`[2] с открытым исходным кодом, в которую входит шаблон класса `Poly`[3], позволяющий реализовать полиморфизм на основе стирания типов;
- библиотека `Dyno`[4, 5], которую разработал Луи Дионн (Louis Dionne), гуру шаблонов C++, также позволяющая реализовать полиморфизм на основе стирания типов.

10.17.2. Методы статического полиморфизма

В современном C++ есть важная тенденция: выполнять как можно больше работы на этапе компиляции, чтобы повысить надежность проверки типов и быстродействие программ[6, 7]. В главах 13 и 14 мы рассмотрим некоторые шаблоны классов стандартной библиотеки, а в главе 15 — программирование пользовательских шаблонов. Программисты часто используют шаблоны для того, чтобы генерировать на этапе компиляции код, способный работать со многими типами данных, — он генерируется для каждого типа отдельно. Пользовательские шаблоны и шаблоны функций используются для реализации статического полиморфизма. Благодаря шаблонам, компилятор может выявлять ошибки еще

---

[1] Артур О'Двайр (Arthur O'Dwyer), «What Is Type Erasure?». Был доступен: 15 сентября 2023 г. https://quuxplusone.github.io/blog/2019/03/18/what-is-type-erasure.

[2] «Folly Open-Source Library». Был доступен: 15 сентября 2023 г. https://github.com/facebook/folly.

[3] «folly/Poly.h». Был доступен: 15 сентября 2023 г. https://github.com/facebook/folly/blob/master/folly/docs/Poly.md.

[4] «Dyno: Runtime Polymorphism Done Right». Был доступен: 15 сентября 2023 г. https://github.com/ldionne/dyno.

[5] Луи Дионн, «Runtime Polymorphism: Back to the Basics». Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=gVGtNFg4ay0.

[6] «C++ Core Guidelines — Per: Performance». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-performance.

[7] «Big Picture Issues — What's the Big Deal with Generic Programming?». Был доступен: 15 сентября 2023 г. https://isocpp.org/wiki/faq/big-picture#generic-paradigm.

до начала выполнения программы и оптимизировать машинный код для повышения быстродействия[1]. В этом разделе мы кратко упомянем несколько методов статического полиморфизма.

Неудавшаяся подстановка не является ошибкой (SFINAE)

Шаблоны поддерживают статический полиморфизм: вы можете генерировать по шаблону класса или функции код, специализированный для работы с одним или несколькими заданными типами. Шаблоны часто предъявляют определенные требования к типам, обязательные для генерации кода. Рассмотрим наш шаблон функции `maximum` из раздела 5.16. Он сравнивает полученные аргументы оператором >, поэтому совместим только с теми типами, к которым применим этот оператор. Если мы предоставим этому шаблону функции аргументы, несовместимые с оператором >, компилятор сообщит об ошибке. **SFINAE (substitution failure is not an error — неудавшаяся подстановка не является ошибкой)**[2,3,4] — это термин, означающий, что компилятор отбрасывает неподходящие варианты кода в процессе выбора вызываемой функции. Если компилятор нашел подходящий вариант, код считается корректным. SFINAE не позволяет компилятору выдавать многочисленные сообщения об ошибках с первой же неудачной попытки специализации шаблона. В главе 15 мы рассмотрим SFINAE в контексте концептов C++20[5,6].

Концепты C++20

Концепты C++20[7,8,9] упрощают программирование с шаблонами, позволяя вам задавать ограничения для типов, которые могут использоваться для специализации шаблонов. Компилятор может проверить, удовлетворяет ли тип заданным для

---

[1] «Compile Time vs Run Time Polymorphism in C++ Advantages/Disadvantages». Был доступен: 15 сентября 2023 г. https://stackoverflow.com/questions/16875989/compile-time-vs-run-timepolymorphism-in-c-advantages-disadvantages.

[2] «Substitution Failure Is Not an Error». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Substitution_failure_is_not_an_error.

[3] Бартломей Филипек, «Notes on C++ SFINAE, Modern C++ and C++20 Concepts». Был доступен: 15 сентября 2023 г. https://www.bfilipek.com/2016/02/notes-on-c-sfinae.html.

[4] Давид Вандевурд (Daveed Vandevoorde) и Николай М. Хосуттис (Nicolai M. Josuttis), «C++ Templates: The Complete Guide». Издательство Addison-Wesley Professional, 2002.

[5] Мариус Банчила (Marius Bancila), «Concepts versus SFINAE-Based Constraints». Был доступен: 15 сентября 2023 г. https://mariusbancila.ro/blog/2019/10/04/concepts-versus-sfinae-basedconstraints.

[6] Бартломей Филипек, «Notes on C++ SFINAE, Modern C++ and C++20 Concepts».

[7] Саар Раз (Saar Raz), «C++20 Concepts: A Day in the Life». Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=qawSiMIXtE4.

[8] «Constraints and Concepts». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/language/constraints.

[9] «Concepts library». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/concepts.

шаблона ограничениям, прежде чем использовать шаблон для генерации кода. В главе 15 мы покажем возможности концептов C++20[1].

Диспетчеризация тегов

Диспетчеризация тегов (tag dispatch)[2] — это основанный на шаблонах метод, в котором компилятор выбирает для вызова нужную версию перегруженной функции, учитывая не только параметры типов шаблона, но и свойства этих типов. Бьерн Страуструп в своей статье «Concepts: The Future of Generic Programming» называет свойства типов «концептами», а диспетчеризацию тегов — перегрузкой концептов, или перегрузкой на основе концептов[3]. Затем он рассказывает о том, как заменить диспетчеризацию тегов введенными в C++20 концептами.

10.17.3. Другие концепции полиморфизма

Вас также могут заинтересовать следующие методы:

- **Двойная диспетчеризация (double dispatch)**[4,5], также называемая **шаблоном посетителя (visitor pattern)**[6], — это метод динамического полиморфизма, основанный на том, что нужная функция выбирается в зависимости от актуальных типов двух объектов в момент вызова. Двойную диспетчеризацию можно реализовать с помощью `std::variant` и `std::visit`[7].
- В презентации «Dynamic Polymorphism with Metaclasses and Code Injection» (CPPCON 2020) Сай Бренд, эксперт Microsoft по разработке языка C++, рассказывает о передовых методах динамического полиморфизма, использующих планируемые нововведения в версиях C++23 и C++26[8].

---

[1] Мариус Банчила (Marius Bancila), «Concepts versus SFINAE-based constraints».

[2] «Generic Programming: Tag Dispatching». Был доступен: 15 сентября 2023 г. https://www.boost.org/community/generic_programming.html#tag_dispatching.

[3] Бьерн Страуструп, «Concepts: The Future of Generic Programming (Section 6)». Был доступен: 15 сентября 2023 г. https://www.stroustrup.com/good_concepts.pdf.

[4] «Double Dispatch». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Double_dispatch.

[5] Барат Каннан (Barath Kannan), «Generalised Double Dispatch». Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=nNqiBasCab4.

[6] «Visitor Pattern». 11 января 2022 г. https://en.wikipedia.org/wiki/Visitor_pattern.

[7] Вишал Човатия (Vishal Chovatiya), «Double Dispatch in C++: Recover Original Type of the Object Pointed by Base Class Pointer». Был доступен: 15 сентября 2023 г. http://www.vishalchovatiya.com/double-dispatch-in-cpp.

[8] Сай Бренд (Sy Brand), «Dynamic Polymorphism with Metaclasses and Code Injection». Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=8c6BAQcYF_E.

10.18. Итоги

В этой главе мы продолжили тему объектно-ориентированного программирования (ООП), рассказав об основах наследования и динамического полиморфизма. Теперь вы умеете создавать производные классы, наследующие элементы базовых классов, а затем настраивать и дополнять их. Мы провели различие между отношением композиции «содержит» («has-a») и отношением наследования «является» («is-a»).

На примере иерархий наследования мы объяснили и продемонстрировали принципы динамического полиморфизма. Вы научились писать программы, которые единообразно обрабатывают объекты разных классов, входящих в одну иерархию, как если бы все они были объектами базового класса иерархии. Вы научились обращаться к этим объектам через указатели и ссылки базового класса.

Мы показали, что динамический полиморфизм помогает разрабатывать и писать большие программы, которые легко расширять, добавляя новые классы почти без изменений в основных частях кода. Мы детально объяснили, как динамический полиморфизм может быть реализован «под капотом» компилятора C++.

Мы представили вам идиому невиртуального интерфейса (NVI). Согласно этой идиоме каждая функция базового класса может быть либо открытой невиртуальной функцией, которую клиентский код может вызывать для выполнения задачи, либо скрытой (или защищенной) виртуальной функцией, которую могут переопределять производные классы. Вы узнали, что при применении этой идиомы виртуальные функции представляют собой внутренние детали реализации, скрытые от клиентского кода, что упрощает поддержку и модификацию программ.

Мы рассказали о наследовании реализации, а затем отметили сложность поддержки и модификации такого кода, подтвержденную многолетней практикой. Затем мы доработали пример программы для расчета зарплаты, применив наследование интерфейса, чтобы базовый класс содержал только чистые виртуальные функции, реализуемые в производных конкретных классах. Этот пример был основан на композиции и внедрении зависимостей: мы включили в класс указатель на объект, от которого зависит поведение объектов этого класса. Затем мы объяснили, как этот подход, основанный на интерфейсах, облегчает модификацию программ.

Далее мы рассмотрели еще один метод реализации динамического полиморфизма, позволяющий обрабатывать объекты не связанных наследованием классов с помощью утиной типизации, шаблона класса `std::variant` и функции `std::visit`.

Мы продемонстрировали множественное наследование, обсудили его потенциальные проблемы и показали, как они решаются виртуальным наследованием. Мы представили вам спецификатор `protected`, дающий функциям и дружественным классам производного класса доступ к защищенным элементам базового класса. Мы сравнили три типа наследования: открытое, защищенное и скрытое — и их влияние на доступность элементов базового класса в производном классе.

Наконец, мы упомянули другие методы динамического полиморфизма и несколько методов статического полиморфизма на основе шаблонов, предоставив ссылки на ресурсы для разработчиков, желающих углубиться в эту тему.

Изучив эту главу, вы освоили принципы наследования и динамического полиморфизма на основе виртуальных функций. Теперь вы сможете лучше оценить другие идиомы и методы полиморфизма, которые могут упрощать модификацию программ и ускорять их работу. Некоторые из этих современных идиом мы рассмотрим в последующих главах.

В главе 11 мы продолжим тему объектно-ориентированного программирования. Мы расскажем о перегрузке операторов, которая позволяет стандартным операторам языка C++ работать с объектами пользовательских типов. Например, вы узнаете, как перегрузить оператор << для вывода всех элементов массива без явного применения инструкции цикла. Вы будете использовать умные указатели, чтобы управлять динамически выделяемой памятью, гарантируя ее освобождение после использования. Мы представим вам ряд специальных функций класса и объясним, как их правильно применять. Мы рассмотрим семантику перемещения, которая позволяет передавать ресурсы (прежде всего динамически выделяемую память) вышедших из области видимости объектов другим объектам, чтобы экономить память и повышать быстродействие.

11

Перегрузка операторов, семантика копирования и перемещения, умные указатели

В этой главе вы:

- познакомитесь с перегруженными операторами класса `string`;
- изучите основы перегрузки операторов, которая поможет вам создавать необходимые классы;
- познакомитесь со специальными функциями классов и узнаете, когда их надо реализовывать для пользовательских типов;
- узнаете, когда объекты надо перемещать, а когда копировать;
- научитесь использовать ссылки *rvalue* и семантику перемещения, чтобы повышать быстродействие программ, не допуская ненужного копирования объектов, выходящих из области видимости;
- узнаете, почему надо избегать управления динамической памятью с помощью операторов `new` и `delete`;
- оцените удобство умных указателей, автоматизирующих управление динамической памятью;
- создадите класс `myArray`, определяющий пять специальных функций для поддержки семантики копирования и перемещения и перегружающий много унарных и бинарных операторов;
- познакомитесь с оператором трехстороннего сравнения (`<=>`), введенным в C++20;
- научитесь преобразовывать типы объектов;
- с помощью спецификатора `explicit` решите проблему неявных преобразований типов конструкторами и операторами;
- испытаете озарение, когда полностью осознаете элегантность и красоту концепции класса.

11.1. Введение

В этой главе рассматривается перегрузка операторов, позволяющая стандартным операторам языка C++ работать с объектами пользовательских классов. Одним из примеров такой перегрузки в стандарте C++ является оператор `<<`, который используется как:

- оператор вставки в поток;
- оператор побитового сдвига влево.

Аналогично оператор `>>` используется как:

- оператор извлечения из потока;
- оператор побитового сдвига вправо.

Вы уже работали с перегруженными операторами. Некоторые из них встроены в ядро языка C++. Например, оператор `+` обрабатывает данные основных типов по-разному, в зависимости от контекста выполняемых операций над целыми числами, числами с плавающей точкой или указателями. Кроме того, этот оператор перегружен в классе `string`, где выполняет конкатенацию строк.

Вы можете перегружать большинство операторов и использовать их с объектами пользовательских классов. Компилятор будет генерировать соответствующий код, основываясь на типах операндов. Действия, выполняемые перегруженными операторами, можно выполнять и явными вызовами функций, но чаще всего аналогичный код с операторами выглядит более удобно и естественно.

Перегруженные операторы класса string

В разделе 2.7 мы познакомили вас с классом `string` стандартной библиотеки и продемонстрировали некоторые его возможности, например конкатенацию строк оператором `+`. Здесь мы рассмотрим еще несколько операторов, перегруженных в классе `string`, чтобы показать, как удобно они реализованы в этом ключевом классе стандартной библиотеки. Затем мы перейдем к перегрузке операторов в пользовательских классах.

Управление динамической памятью и умные указатели

Вы познакомитесь с управлением динамической памятью, которое позволяет программе получать дополнительную память для объектов во время выполнения (а не заранее, при компиляции) и освобождать эту память, когда она больше не нужна, чтобы ее можно было использовать для других целей. Мы обсудим типичные проблемы с динамически выделяемой памятью, в том числе **утечку памяти (memory leak)**, возникающую, когда программист забывает освободить уже ненужную ему память. Вы познакомитесь с **умными указателями (smart pointers)**, которые автоматически освобождают динамически выделяемую память, избавляя вас от лишних забот. Как вы увидите, в сочетании с идиомой RAII (Resource Acquisition Is Initialization — «получение ресурсов есть инициализация») умные указатели предотвращают утечку памяти.

Класс myArray

Далее мы перейдем к одному из важнейших учебных примеров этой книги. Мы создадим пользовательский класс `myArray`, использующий перегруженные операторы и другие возможности C++ для решения проблем традиционных массивов в стиле C. Мы введем и реализуем пять специальных функций, которые вы можете определять в каждом своем классе: копирующий конструктор, оператор копирующего присваивания, перемещающий конструктор, оператор перемещающего присваивания и деструктор. В некоторых случаях вы можете добавлять в класс еще одну специальную функцию: конструктор по умолчанию. Мы рассмотрим семантику копирования, а также семантику перемещения — синтаксис, который помогает компилятору понять, что можно перемещать ресурсы из одного объекта в другой, избегая ненужного ресурсоемкого копирования.

Класс `myArray` будет использовать умные указатели в сочетании с RAII, а также перегруженные унарные и двоичные операторы, в том числе:

- `=` (оператор присваивания);
- `==` (оператор равенства);
- `!=` (оператор неравенства);
- `[]` (оператор индекса);
- `++` (оператор инкремента);
- `+=` (оператор сложения с присваиванием);
- `>>` (оператор извлечения из потока);
- `<<` (оператор вставки в поток).

Мы покажем, как определить конвертирующий оператор, то есть оператор, выполняющий преобразования между типами данных. Он будет преобразовывать (конвертировать) тип `myArray` в значение `true` или `false`, указывающее, содержит ли объект класса `myArray` какие-либо элементы или является пустым.

Многие из наших читателей говорили, что работа над классом `myArray` — это «озарение», которое помогает по-настоящему оценить, что такое классы и объектная технология. Как только вы освоите класс `myArray`, вы полностью поймете суть объектной технологии: творческое создание, использование и повторное использование уникальных классов, обмен ими с коллегами и, возможно, со всем сообществом открытого исходного кода C++.

Оператор трехстороннего сравнения (<=>)

Мы расскажем о возможностях введенного в C++20 оператора трехстороннего сравнения `<=>`, который иногда называют «звездолетом» («spaceship»)[1].

[1] Название «звездолет» придумал Рэндал Л. Шварц, когда рассказывал студентам об аналогичном операторе в курсе программирования на Perl. Оператор напомнил ему космический корабль в ранней видеоигре. https://groups.google.com/a/dartlang.org/g/misc/c/WS5xftltpl4/m/jcIttrMq8agJ?pli=1.

Конвертирующие операторы

Мы подробно обсудим перегруженные конвертирующие операторы и конвертирующие конструкторы. В частности, мы покажем, как неявное преобразование одних типов в другие приводит к скрытым проблемам. Затем мы используем ключевое слово `explicit` для предотвращения этих проблем.

11.2. Перегруженные операторы класса string стандартной библиотеки

В главе 8 был представлен класс `string`. Код, показанный на илл. 11.1, демонстрирует многие перегруженные операторы этого класса и некоторые другие его функции, включая `empty`, `substr` и `at`:

- функция `empty` определяет, является ли строка пустой;
- функция `substr` (подстрока) возвращает указанную часть строки;
- функция `at` возвращает символ с указанным индексом в строке (после проверки нахождения индекса в допустимом диапазоне).

Для удобства обсуждения мы разделили этот пример на небольшие фрагменты, за которыми следуют их выходные данные.

```
1 // fig11_01.cpp
2 // Тестирование класса string стандартной библиотеки
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <string>
6 #include <string_view>
7
8 int main() {
```

Илл. 11.1 | Тестирование класса string стандартной библиотеки

Создание объектов string и string_view, отображение их содержимого с помощью cout и оператора <<

В строках 9–12 создаются три объекта `string` и один объект `string_view`:

- объект `s1` инициализируется литералом `"happy"`;
- объект `s2` инициализируется литералом `" birthday"`;
- объект `s3` использует конструктор класса `string` по умолчанию для создания пустой строки;
- объект `string_view` с именем `v` инициализируется литералом `"hello"`.

В строках 15–16 эти объекты отображаются с помощью `cout` и оператора `<<`, перегруженного в классах `string` и `string_view` для вывода данных на экран.

```
 9    std::string s1{"happy"}; // Инициализируем объект string
10    std::string s2{" birthday"}; // Инициализируем объект string
11    std::string s3; // Создаем пустой объект string
12    std::string_view v{"hello"}; // Инициализируем объект string_view
13
14    // Выводим содержимое всех четырех объектов
15    std::cout << "s1: \"" << s1 << "\"; s2: \"" << s2
16       << "\"; s3: \"" << s3 << "\"; v: \"" << v << "\"\n\n";
17
```

```
s1: "happy"; s2: " birthday"; s3: ""; v: "hello"
```

Сравнение строковых объектов операторами равенства и сравнения

В строках 19–25 выводятся результаты сравнения строки `s2` со строкой `s1` перегруженными операторами равенства и сравнения класса `string`. Они выполняют лексикографические сравнения, используя числовые значения символов в каждой строке (см. приложение Б). Когда вы вызываете функцию `format` для преобразования значения `bool` в значение `string`, она выводит строку `"true"` или `"false"`.

```
18    // Перегруженные операторы равенства и сравнения
19    std::cout << "The results of comparing s2 and s1:\n"
20       << fmt::format("s2 == s1: {}\n", s2 == s1)
21       << fmt::format("s2 != s1: {}\n", s2 != s1)
22       << fmt::format("s2 > s1: {}\n", s2 > s1)
23       << fmt::format("s2 < s1: {}\n", s2 < s1)
24       << fmt::format("s2 >= s1: {}\n", s2 >= s1)
25       << fmt::format("s2 <= s1: {}\n\n", s2 <= s1);
26
```

```
The results of comparing s2 and s1:
s2 == s1: false
s2 != s1: true
s2 > s1: false
s2 < s1: true
s2 >= s1: false
s2 <= s1: true
```

Функция empty класса string

В строке 30 вызывается функция `empty`, которая возвращает `true`, если строка пуста; в противном случае она возвращает `false`. Объект `s3` был инициализирован конструктором по умолчанию, поэтому он содержит пустую строку.

```
27    // Функция empty класса string
28    std::cout << "Testing s3.empty():\n";
29
30    if (s3.empty()) {
31       std::cout << "s3 is empty; assigning s1 to s3;\n";
32       s3 = s1; // Присваиваем значение объекта s1 объекту s3
33       std::cout << fmt::format("s3 is \"{}\"\n\n", s3);
```

Окончание ⇨

```
34    }
35
```

```
Testing s3.empty():
s3 is empty; assigning s1 to s3;
s3 is "happy"
```

Оператор копирующего присваивания класса string

В строке 32 демонстрируется перегруженный **оператор копирующего присваивания (copy assignment operator)** класса `string`, который присваивает значение объекта `s1` объекту `s3`. В строке 33 выводится значение объекта `s3` для подтверждения корректности выполненного присваивания.

Конкатенация строк и строковые объектные литералы (C++14)

В строке 37 демонстрируется перегруженный составной оператор `+=`, выполняющий конкатенацию строк с присваиванием. В этом случае содержимое строки `s2` добавляется в конец строки `s1`, при этом изменяется значение `s1`. Затем в строке 38 выводится обновленное значение `s1`. В строке 41 демонстрируется добавление строки в стиле C в конец строки типа `string` составным оператором `+=`. В строке 42 выводится результат. Аналогично в строке 46 строка `s1` объединяется со строковым объектным литералом (C++14). Строковый объектный литерал обозначается буквой `s` после закрывающей кавычки строкового литерала, например:

```
", have a great day!"s
```

Этот литерал фактически вызывает функцию стандартной библиотеки C++, которая возвращает объект `string`, содержащий символы литерала. В строках 47–48 отображается новое значение `s1`.

```
36    // Перегруженный оператор конкатенации с присваиванием
37    s1 += s2; // Конкатенация двух строк
38    std::cout << fmt::format("s1 += s2 yields s1 = {}\n\n", s1);
39
40    // Конкатенация строки типа string со строкой в стиле C
41    s1 += " to you";
42    std::cout << fmt::format("s1 += \" to you\" yields s1 = {}\n\n", s1);
43
44    // Конкатенация строки типа string со строковым объектным литералом
45    using namespace std::string_literals;
46    s1 += ", have a great day!"s; // s после " - строковый объектный литерал
47    std::cout << fmt::format(
48       "s1 += \", have a great day!\"s yields\ns1 = {}\n\n", s1);
49
```

```
s1 += s2 yields s1 = happy birthday

s1 += " to you" yields s1 = happy birthday to you

s1 += ", have a great day!"s yields
s1 = happy birthday to you, have a great day!
```

Функция substr класса string

Класс `string` предоставляет функцию `substr` (строки 53 и 58), возвращающую подстроку (часть строки) объекта `string`, для которого она вызывается. Вызов в строке 53 создает подстроку объекта `s1` длиной `14` символов, начинающуюся с нулевой позиции. Вызов в строке 58 создает подстроку, начинающуюся с `15`-й позиции объекта `s1`. Если второй аргумент не указан, функция `substr` возвращает всю оставшуюся часть строки, для которой она вызывается.

```
50    // Функция substr класса string
51    std::cout << fmt::format("{} {}\n{}\n\n",
52       "The substring of s1 starting at location 0 for",
53       "14 characters, s1.substr(0, 14), is:", s1.substr(0, 14));
54
55    // Опция функции substr "до конца строки"
56    std::cout << fmt::format("{} {}\n{}\n\n",
57       "The substring of s1 starting at",
58       "location 15, s1.substr(15), is:", s1.substr(15));
59
```

```
The substring of s1 starting at location 0 for 14 characters,
s1.substr(0, 14), is:
happy birthday

The substring of s1 starting at location 15, s1.substr(15), is:
to you, have a great day!
```

Копирующий конструктор класса string

В строке 61 создается строковый объект `s4` и инициализируется копией объекта `s1`. При этом вызывается копирующий конструктор класса `string`, который копирует содержимое `s1` в новый объект `s4`. Вы научитесь определять копирующий конструктор для пользовательского класса в разделе 11.6.

```
60    // Копирующий конструктор
61    std::string s4{s1};
62    std::cout << fmt::format("s4 = {}\n\n", s4);
63
```

```
s4 = happy birthday to you, have a great day!
```

Самоприсваивание оператором копирующего присваивания класса string

В строке 66 используется перегруженный оператор копирующего присваивания (`=`) класса `string` для демонстрации корректности **самоприсваивания (self-assignment)**. После выполнения этой инструкции строка `s4` по-прежнему имеет исходное значение. Далее в этой главе, когда мы создадим класс `myArray`, мы покажем, что надо очень осторожно обращаться с самоприсваиванием при работе с объектами, управляющими собственной памятью, и рассмотрим пути решения соответствующих проблем.

```
64    // Самоприсваивание перегруженным оператором копирующего присваивания
65    std::cout << "assigning s4 to s4\n";
66    s4 = s4;
67    std::cout << fmt::format("s4 = {}\n\n", s4);
68
```

```
assigning s4 to s4
s4 = happy birthday to you, have a great day!
```

Инициализация строки объектом string_view

В строке 71 демонстрируется конструктор класса `string`, получающий объект `string_view` и в данном случае копирующий символьные данные из объекта `v` (строка 12) в новую строку `s5`.

```
69    // Конструктор класса string с аргументом string_view
70    std::cout << "initializing s5 with string_view v\n";
71    std::string s5{v};
72    std::cout << fmt::format("s5 is {}\n\n", s5);
73
```

```
initializing s5 with string_view v
s5 is hello
```

Оператор [] класса string

В строках 75–76 перегруженный оператор [] используется в операциях присваивания, создавая значения *lvalue* для замены символов в `s1`. В строках 77–78 выводится новое значение `s1`. Оператор [] возвращает символ в указанной позиции как изменяемое значение *lvalue* или неизменяемое (константное) значение *lvalue*, в зависимости от контекста использования в выражении. Например:

- если оператор [] используется в строке, не являющейся константой, функция возвращает изменяемое значение *lvalue*, которое можно использовать слева от оператора присваивания (=), чтобы присвоить новое значение в этом месте строки, как показано в строках 77–78;
- если оператор [] используется в строке, являющейся константой, функция возвращает неизменяемое значение *lvalue*, которое можно использовать для получения, но не для изменения значения в этом месте строки.

Перегруженный оператор [] не выполняет проверку границ. Вы должны проверять, что инструкции с этим оператором не обращаются к элементам за границами строки.

```
74    // Перегруженный оператор индекса создает lvalue
75    s1[0] = 'H';
76    s1[6] = 'B';
77    std::cout << fmt::format("{}:\n{}\n\n",
78       "after s1[0] = 'H' and s1[6] = 'B', s1 is", s1);
79
```

```
after s1[0] = 'H' and s1[6] = 'B', s1 is:
Happy Birthday to you, have a great day!
```

Функция at класса string

Функция `at` класса `string` выдает исключение, если ее аргументом является недопустимый индекс, выходящий за границы строки. Если индекс допустимый, функция `at` возвращает символ в указанной позиции как изменяемое значение *lvalue* или неизменяемое (константное) значение *lvalue*, в зависимости от контекста вызова. В строке 83 демонстрируется вызов функции `at` с недопустимым индексом, вызывающим исключение `out_of_range`. Здесь мы показываем сообщение об ошибке, созданное компилятором GNU C++.

```
80    // Тестируем недопустимый индекс в функции at класса string
81    try {
82       std::cout << "Attempt to assign 'd' to s1.at(100) yields:\n";
83       s1.at(100) = 'd'; // Ошибка: недопустимый индекс!
84    }
85    catch (const std::out_of_range& ex) {
86       std::cout << fmt::format("An exception occurred: {}\n", ex.what());
87    }
88 }
```

```
Attempt to assign 'd' to s1.at(100) yields:
An exception occurred: basic_string::at: __n (which is 100) >= this->size()
(which is 40)
```

11.3. Основные принципы перегрузки операторов

Как было показано на илл. 11.1, перегруженные операторы класса `string` стандартной библиотеки представляют собой лаконичную форму записи операций со строковыми объектами. Вы также можете использовать операторы с пользовательскими, то есть вашими собственными, типами. C++ позволяет перегружать большинство существующих операторов, определяя **операторные функции (operator functions)**. Как только вы определите операторную функцию для какого-либо оператора и вашего пользовательского класса, этот оператор будет выполнять нужные вам операции с объектами этого класса.

11.3.1. Перегрузка операторов не происходит автоматически

Вы должны написать операторные функции, выполняющие нужные операции. Для перегрузки оператора требуется определение нестатической функции пользовательского класса или функции, которая не является элементом класса. Операторная функция начинается с ключевого слова `operator`, за которым следует символ перегружаемого оператора. Например, функция с именем `operator+` будет перегружать оператор сложения (`+`). Когда операторы перегружаются как функции пользовательских классов, они должны быть нестатическими. Они вызываются для объекта класса и работают с этим объектом.

11.3.2. Операторы, которые не могут быть перегружены

Большинство операторов C++ можно перегружать, кроме нескольких[1]:

- . (оператор «точка»);
- * (оператор указателя на элемент; см. раздел 20.6);
- :: (оператор разрешения области видимости);
- ?: (тернарный условный оператор; возможность его перегрузки может быть реализована в будущем)[2].

11.3.3. Операторы, которые не нужно перегружать

Не перегружайте эти три оператора, работающие с объектами любого нового класса по умолчанию:

- Оператор присваивания (=) может быть использован с большинством классов для выполнения последовательных присваиваний элементов данных. По умолчанию этот оператор присваивает каждый элемент данных из объекта «источник» (справа от оператора) объекту «цель» (слева от оператора). Как вы увидите в разделе 11.6.6, такие операции могут быть опасными с классами, содержащими указатели в качестве элементов данных. В этом случае вы либо явно перегружаете оператор присваивания, либо явно запрещаете компилятору определять оператор присваивания по умолчанию. Это также верно для оператора, перемещающего присваивания (C++11), который мы обсудим в разделе 11.6.
- Оператор взятия адреса (&), возвращающий указатель на объект.
- Оператор «запятая», вычисляющий выражение слева от него, затем выражение справа от него и возвращающий значение последнего выражения. Хотя этот оператор может быть перегружен, на практике это не работает[3].

11.3.4. Правила и ограничения перегрузки операторов

Когда вы готовитесь перегружать операторы для работы с пользовательскими классами, вам надо учитывать несколько правил и ограничений:

- Перегрузка не изменяет приоритет оператора. Круглые скобки могут быть использованы для изменения порядка вычисления перегруженных операторов в выражении.

---

[1] Оператор взятия адреса (&), оператор «запятая» (,) оператор && и оператор || теоретически можно перегружать, но лучше этого не делать, чтобы избежать скрытых ошибок. Подробнее: https://isocpp.org/wiki/faq/operator-overloading. Был доступен: 15 сентября 2023 г.

[2] Маттиас Крец (Matthias Kretz), «Making Operator ?: Overloadable», 7 октября 2019 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p0917.

[3] «Operator Overloading». Был доступен: 15 сентября 2023 г. https://isocpp.org/wiki/faq/operatoroverloading.

- Перегрузка не изменяет группировку (ассоциативность) оператора. Если оператор обычно группируется слева направо, его перегруженные версии группируются таким же образом.
- Перегрузка не изменяет «арность» оператора (количество операндов, которые он может принять). Перегруженные унарные операторы остаются унарными, а перегруженные бинарные операторы остаются бинарными. Единственный тернарный оператор `?:` не может быть перегружен. Операторы `&`, `*`, `+` и `-` имеют унарную и бинарную версии, которые могут быть перегружены отдельно.
- Перегружать можно только существующие операторы. Вы не можете создавать новые.
- Вы не можете перегружать операторы для изменения принципа работы со значениями основных типов. Например, вы не можете перегрузить оператор `+` для вычитания целых чисел.
- Перегрузка оператора работает только с объектами пользовательских типов или с сочетанием объекта пользовательского типа и объекта базового типа.
- Схожие операторы, например `+` и `+=`, как правило, должны перегружаться отдельно. В C++20, если определить оператор `==` для базового класса, C++ автоматически предоставит перегруженный оператор `!=`, который будет возвращать значение, противоположное оператору `==`.
- При перегрузке операторов `()`, `[]`, `->` и `=` функция перегрузки должна быть объявлена как элемент класса. Позже вы увидите, что это требуется, когда левый операнд должен иметь тип пользовательского класса. Функции перегрузки для всех других операторов могут быть объявлены как внутри класса, так и вне класса.

Вы должны перегружать операторы для пользовательских классов таким образом, чтобы логика их работы была как можно ближе к логике работы с основными типами. Избегайте чрезмерного или нелогичного использования перегрузки операторов, чтобы не делать ваш исходный код загадочным и трудным для понимания.

11.4. Управление динамической памятью с помощью операторов new и delete

Вы можете выделять и освобождать память в программе для объектов и массивов любого встроенного или определяемого пользователем типа. Это называется управлением динамической памятью. В главе 7 мы познакомили вас с указателями и показали различные методы «старого стиля», которые вы можете встретить в устаревшем коде. Затем мы рассмотрели более совершенные методы современного C++. Здесь мы сделаем то же самое. На протяжении десятилетий управление динамической памятью в C++ выполнялось операторами `new` и `delete`. Но теперь в C++ Core Guidelines рекомендуется не использовать эти

операторы напрямую[1,2]. Вы будете встречать их в устаревшем коде C++, поэтому мы рассмотрим их в этом разделе. В разделе 11.5 мы обсудим современные методы управления динамической памятью в C++, а затем применим их на практике в разделе 11.6, посвященном классу `myArray`.

Старая школа: операторы new и delete

Вы можете использовать оператор `new` для динамического резервирования (выделения) точного объема памяти, необходимого для хранения объекта или традиционного массива. Объект или традиционный массив создается в **свободном пуле (free store)** — это область памяти, предоставленная каждой программе для хранения динамически размещаемых объектов. Когда память выделена, вы можете получить к ней доступ через указатель, возвращаемый оператором `new`. Когда вам больше не нужна эта память, вы можете вернуть ее в свободный пул. Оператор `delete` освобождает память, то есть возвращает ее операционной системе. Далее эта память может повторно использоваться в будущих операциях.

Выделение динамической памяти оператором new

Рассмотрим следующую инструкцию:

```
Time* timePtr{new Time{}};
```

Оператор `new` выделяет память для объекта класса `Time`, вызывая конструктор по умолчанию для инициализации и возвращая `Time*` — указатель с типом, указанным справа от оператора `new`. Эта инструкция вызывает конструктор класса `Time` по умолчанию, потому что мы не предоставили аргументы. Если оператор `new` не может найти в памяти достаточно места для объекта, он выдает исключение `bad_alloc`. В главе 12 показано, как работать с исключениями оператора `new`.

Освобождение динамической памяти оператором delete

Чтобы уничтожить динамически размещенный объект и освободить занимаемое им место, используйте оператор `delete`:

```
delete timePtr;
```

Эта инструкция вызывает деструктор объекта, на который указывает `timePtr`, и освобождает память объекта, возвращая ее в свободный пул. Если не освобо-

---

[1] C++ Core Guidelines, «R.11: Avoid Calling new and delete Explicitly». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rr-newdelete.

[2] Операторы new и delete можно перегружать. Если вы перегружаете оператор new, вы должны перегрузить оператор delete в той же области видимости, чтобы избежать скрытых ошибок управления динамической памятью. Перегрузка операторов new и delete обычно нужна для точного контроля над выделением и освобождением памяти. Ее можно использовать, например, для того, чтобы заранее выделить пул памяти, а потом размещать в нем объекты, снижая связанную с выделением памяти нагрузку во время выполнения программы. Для этого у операторов new и delete есть специальные версии: «placement new» и «placement delete», см. https://en.wikipedia.org/wiki/Placement_syntax.

дить динамически выделяемую память, когда она больше не нужна, это может привести к утечке памяти, которая в конечном итоге приведет к преждевременной нехватке памяти в операционной системе. Иногда эта проблема может расти как снежный ком. Если в памяти, которую вы забыли освободить, находятся объекты, управляющие другими ресурсами, деструкторы этих объектов не будут вызываться для освобождения этих ресурсов, что приведет к дополнительным утечкам памяти.

Не освобождайте память, которая не была выделена с помощью оператора `new`. Это приводит к неопределенному поведению. После освобождения выделенной памяти убедитесь, что вы не собираетесь освобождать ее повторно, так как это обычно приводит к сбою программы. Один из способов защититься от подобных сбоев — сразу присвоить указателю значение `nullptr`; освобождение такого указателя не повлечет за собой никаких действий системы.

Инициализация динамически размещаемого объекта

Вы можете инициализировать новый динамически размещаемый объект аргументами конструктора, например:

```
Time* timePtr{new Time{12, 45, 0}};
```

Эта инструкция инициализирует новый объект класса `Time` значением времени 12:45:00 PM и присваивает адрес этого объекта указателю `timePtr`.

Выделение памяти для традиционного массива оператором new[]

Вы можете использовать оператор `new[]` для выделения динамической памяти традиционным массивам в стиле C. Следующая инструкция выделяет память для 10-элементного массива целых чисел:

```
int* gradesArray{new int[10]{}};
```

Эта инструкция направляет указатель `gradesArray` на первый элемент динамически размещаемого массива. Пустой скобочный инициализатор, следующий за `new int[10]`, инициализирует элементы массива, присваивая элементам основных типов значение `0`, типам `bool` — значение `false`, а указателям — `nullptr`. Скобочный инициализатор может также содержать разделенный запятыми список инициализаторов для элементов массива. При инициализации объекта по умолчанию вызывается его конструктор по умолчанию, если он есть. Правила усложняются для объектов без конструкторов по умолчанию. Здесь вы найдете более подробную информацию о правилах инициализации:

https://en.cppreference.com/w/cpp/language/value_initialization

Размер массива, созданного во время компиляции, должен быть указан в виде константы (константного целочисленного выражения). Однако размер динамически размещаемого массива можно указывать в виде любого неотрицательного целочисленного выражения.

Освобождение памяти, выделенной для традиционного массива оператором delete[]

Чтобы освободить память, на которую указывает `gradesArray`, используйте такую инструкцию:

```
delete[] gradesArray;
```

Если указатель ссылается на традиционный массив объектов, эта инструкция сначала вызывает деструкторы каждого объекта в массиве, а затем освобождает всю память массива. Как и в случае с `delete`, оператор `delete[]` для `nullptr` не имеет никакого эффекта.

Использование оператора `delete` вместо `delete[]` для памяти, выделенной оператором `new[]`, приводит к неопределенному поведению. Некоторые компиляторы вызывают деструктор только для первого объекта в массиве. Чтобы гарантировать вызов деструкторов для всех объектов массива, всегда используйте оператор `delete[]` для освобождения памяти, выделенной оператором `new[]`. Далее мы рассмотрим более удобные методы управления динамически выделяемой памятью, которые позволят обойтись без операторов `new` и `delete`.

Если имеется только один указатель на какой-либо блок выделенной памяти, то в случае выхода этого указателя за пределы области видимости или в случае, если вы присвоите ему `nullptr` или другой адрес памяти, происходит утечка памяти.

После освобождения выделенной памяти установите значение указателя равным `nullptr`, чтобы он не указывал на память в свободном пуле. Это гарантирует, что код не сможет непреднамеренно получить доступ к ранее выделенной памяти, приводящий к трудно выявляемым логическим ошибкам.

Оператор for для диапазона значений не работает с массивами в динамически выделяемой памяти

Можно ли использовать инструкцию `for` для диапазона значений (C++11), чтобы обрабатывать элементы в динамически размещаемом массиве? К сожалению, нельзя: такой код не скомпилируется. Компилятор должен уже во время компиляции знать количество элементов, чтобы перебирать их в этом цикле. В качестве обходного пути можно создать объект `span` (C++20), представляющий динамически размещаемый массив и количество его элементов, а затем обработать этот объект `span` с помощью цикла `for` для диапазона значений.

11.5. Современные методы управления динамической памятью в C++: RAII и умные указатели

Общий принцип управления динамической памятью заключается в следующем:

- выделить динамическую память;
- присвоить адрес этой памяти указателю;

- использовать указатель для управления памятью;
- освободить память, когда она больше не нужна.

Если после успешного выделения памяти, но до выполнения `delete` или `delete[]` возникнет исключение, может произойти утечка памяти.

По этой причине в C++ Core Guidelines рекомендуется управлять такими ресурсами, как динамическая память, используя концепцию RAII («получение ресурсов есть инициализация»)[1, 2]. Эта концепция очень проста. Для любого ресурса, который надо вернуть операционной системе после использования, ваша программа должна:

- создать объект как локальную переменную в функции — конструктор объекта должен получить ресурс при инициализации объекта;
- использовать этот объект по мере необходимости;
- освободить ресурс через деструктор объекта, когда вызов функции завершается и объект выходит из области видимости.

11.5.1. Умные указатели

Умные указатели (smart pointers) (C++11) используют RAII для управления динамически выделяемой памятью. Заголовок стандартной библиотеки `<memory>` определяет три типа умных указателей:

- `unique_ptr`
- `shared_ptr`
- `weak_ptr`

Тип `unique_ptr` поддерживает указатель на область динамически выделяемой памяти, которая может принадлежать только одному объекту `unique_ptr`. Когда объект `unique_ptr` выходит за пределы области видимости, его деструктор использует оператор `delete` (или `delete[]` для массива), чтобы освободить ту память, которой управляет указатель `unique_ptr`. Остальная часть раздела 11.5 демонстрирует работу указателей `unique_ptr`, которые мы будем использовать в примерах с классом `myArray`.

11.5.2. Умный указатель unique_ptr

В коде на илл. 11.2 показана работа умного указателя `unique_ptr`, ссылающегося на динамически размещенный объект класса `Integer` (строки 7–22). В демонстрационных целях конструктор и деструктор класса выводят соответствующие сообщения во время их вызова. В строке 29 создается объект `ptr` типа `unique_ptr` и инициализируется указателем на динамически размещаемый объект `Integer`,

---

[1] C++ Core Guidelines, «R: Resource Management». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-resource.

[2] C++ Core Guidelines, «R.1: Manage Resources Automatically Using Resource Handles and RAII (Resource Acquisition Is Initialization)». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rr-raii.

содержащий значение 7. Чтобы инициализировать unique_ptr, в строке 29 вызывается шаблон функции make_unique (C++14), который выделяет динамическую память с помощью оператора new и передает эту память указателю unique_ptr[1]. В этом примере функция make_unique<Integer> возвращает unique_ptr<Integer> — в строке 29 используется ключевое слово auto для автоматического определения типа указателя по типу его инициализатора.

```
1 // fig11_02.cpp
2 // Демонстрация умного указателя unique_ptr
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <memory>
6
7 class Integer {
8 public:
9    // Конструктор
10   Integer(int i) : value{i} {
11      std::cout << fmt::format("Constructor for Integer {}\n", value);
12   }
13
14   // Деструктор
15   ~Integer() {
16      std::cout << fmt::format("Destructor for Integer {}\n", value);
17   }
18
19   int getValue() const {return value;} // Возвращаем значение value
20 private:
21   int value{0};
22 };
23
24 // Используем unique_ptr для работы с объектом Integer
25 int main() {
26   std::cout << "Creating a unique_ptr that points to an Integer\n";
27
28   // Создаем объект unique_ptr и "направляем" его на новый объект Integer
29   auto ptr{std::make_unique<Integer>(7)};
30
31   // Используем указатель unique_ptr для вызова функции класса Integer
32   std::cout << fmt::format("Integer value: {}\n\nMain ends\n",
33      ptr->getValue());
34 }
```

```
Creating a unique_ptr that points to an Integer
Constructor for Integer 7
Integer value: 7

Main ends
Destructor for Integer 7
```

Илл. 11.2 | Демонстрация умного указателя unique_ptr

[1] До C++14 результат выражения с оператором new передавался непосредственно конструктору unique_ptr.

В строке 33 перегруженный оператор `->` класса `unique_ptr` вызывает функцию `getValue` для объекта класса `Integer`, которым управляет указатель `ptr`. Выражение `ptr->getValue()` можно было бы записать и таким образом:

```
(*ptr).getValue()
```

В этой записи перегруженный оператор `*` класса `unique_ptr` используется для разыменования указателя `ptr`, затем с помощью оператора «точка» вызывается функция `getValue` для объекта класса `Integer`.

Поскольку указатель `ptr` является локальной переменной в функции `main`, он уничтожается при ее завершении. Деструктор `unique_ptr` удаляет динамически размещенный объект класса `Integer`, при этом вызывается деструктор объекта. Занятая объектом память освобождается при любом варианте завершения программы: и при достижении инструкции `return`, и при достижении правой закрывающей скобки, и при возникновении исключения.

Важнее всего, что работа с указателями `unique_ptr` предотвращает утечку ресурсов. Предположим, что некоторая функция возвращает обычный указатель, ссылающийся на динамически размещенный объект. Предположим, что код в точке вызова функции, получив указатель, не сможет освободить выделенную память, что приведет к утечке памяти. Но если функция вернет умный указатель `unique_ptr` на этот объект, то занятая объектом память будет автоматически освобождена при вызове деструктора объекта.

11.5.3. Владение объектами при работе с unique_ptr

Только один указатель `unique_ptr` может владеть динамически размещаемым объектом. Поэтому при присваивании одного указателя `unique_ptr` другому передается **владение (ownership)** объектом. Аналогичные действия происходят, когда указатель `unique_ptr` передается в качестве аргумента конструктору другого указателя `unique_ptr`. Эти операции используют оператор перемещающего присваивания и перемещающий конструктор, рассматриваемые в разделе 11.6. Последний из указателей `unique_ptr`, владеющий динамической памятью, освободит ее. Это делает указатели `unique_ptr` идеальным механизмом для того, чтобы возвращать клиентскому коду владение динамически размещаемыми объектами.

11.5.4. Указатели unique_ptr и традиционные массивы

Вы можете использовать указатели `unique_ptr` для управления динамически размещаемыми традиционными массивами, как мы это сделаем в разделе 11.6 в классе `myArray`. Например, в инструкции:

```
auto ptr{make_unique<int[]>(10)};
```

аргументом функции `make_unique` является массив `int[]`. Таким образом, функция `make_unique` динамически размещает традиционный массив с количеством элементов, указанным в его аргументе (`10`). По умолчанию целочисленные элементы инициализируются значением `0`. В этой инструкции используется ключевое слово

auto для автоматического определения типа указателя ptr (unique_ptr<int[]>) по типу его инициализатора.

Указатель unique_ptr, управляющий массивом, предоставляет перегруженный оператор индекса ([]) для доступа к элементам. Например, инструкция

```
ptr[2] = 7;
```

присваивает значение 7 элементу ptr[2] массива int, а следующая инструкция отображает этот элемент:

```
std::cout << ptr[2] << "\n";
```

11.6. Создание класса myArray с перегрузкой операторов

Проектирование классов — это интересная, творческая и сложная работа, цель которой — создание практичного и полезного кода. В этом разделе мы будем называть словом «массив» традиционный массив в стиле C (глава 7). Такие массивы, основанные на указателях, имеют много проблем, в том числе:

- C++ не проверяет выход индекса массива за границы. Программа может легко «вылететь» за границы массива, что почти наверняка приведет к фатальной ошибке во время выполнения, если вы забудете предусмотреть проверку индекса в своем коде.
- Массивы с размерностью n должны использовать значения индекса в диапазоне от 0 до n - 1. Альтернативные диапазоны индексов не допускаются.
- Вы не можете вводить (записывать) массив оператором извлечения из потока (>>) и выводить (считывать) массив оператором вставки в поток (<<). Вам приходится записывать или считывать каждый элемент массива по очереди[1].
- Два массива нельзя сопоставить операторами равенства или сравнения. Имя массива — это просто указатель на начало массива в памяти. Два массива всегда будут находиться в разных ячейках памяти.
- Когда вы передаете массив в функцию, обрабатывающую массив любого размера, вы должны передать еще и дополнительный аргумент: размер массива. Как вы видели в разделе 7.10, объекты span (C++20) помогают решить эту проблему.
- Нельзя присваивать один массив другому операторами присваивания.

В C++ вы можете реализовать работу с массивами более надежно, используя классы и перегруженные операторы, как в шаблонах классов стандартной библиотеки C++ array и vector. В этом разделе мы разработаем пользовательский

[1] В главе 13 вы будете использовать функции стандартной библиотеки C++, чтобы записывать и считывать контейнеры элементов целиком, например контейнеры vector и array.

класс `myArray`, который тоже значительно надежнее традиционных массивов. Внутри класса `myArray` будет использоваться умный указатель `unique_ptr` для управления традиционным массивом с целочисленными значениями в выделенной памяти[1].

Перечислим ценные возможности класса `myArray`:

- Объект класса `myArray` выполняет проверку границ при обращении к нему через оператор индекса (`[]`), чтобы индексы гарантированно находились в пределах границ массива. В противном случае объект `myArray` выдает исключение `out_of_bounds` стандартной библиотеки.
- Весь объект `myArray` можно вводить (записывать) и выводить (считывать) перегруженными операторами извлечения из потока (`>>`) и вставки в поток (`<<`) соответственно, без необходимости циклов в клиентском коде.
- Объекты `myArray` можно сравнивать друг с другом операторами равенства `==` и `!=`. Класс можно будет легко расширить, добавив поддержку операторов сравнения.
- Объект `myArray` знает свой размер, что упрощает его передачу функциям.
- Объекты `myArray` можно присваивать друг другу оператором присваивания.
- Объект `myArray` можно преобразовывать в значение `false` или `true`, показывающее, является ли он пустым или содержит элементы.
- Объект `myArray` предоставляет операторы префиксного и постфиксного инкремента (`++`), прибавляющие `1` к каждому элементу. Мы сможем легко добавить поддержку операторов префиксного и постфиксного декремента (`--`).
- Объект `myArray` предоставляет составной оператор присваивания (`+=`), прибавляющий указанное значение к каждому элементу. Класс можно будет легко расширить, добавив поддержку других составных операторов присваивания: `-=`, `*=`, `/=` и `%=`.

Класс `myArray` будет содержать пять специальных функций, а также умный указатель `unique_ptr` для управления динамически выделяемой памятью. В этом примере мы используем идиому RAII («получение ресурсов есть инициализация») для управления динамическими ресурсами памяти. Конструкторы класса будут выделять память по мере инициализации объектов `myArray`. Деструкторы класса будут предотвращать утечки памяти, освобождая память, когда объекты выходят из области видимости. Наш класс `myArray` не предназначен для замены стандартных шаблонов `array` и `vector` или для имитации их возможностей. В нем демонстрируются ключевые функции языка C++ и функции библиотеки, которые пригодятся вам при создании ваших собственных классов.

---

[1] В этом разделе мы создадим практичный и полезный класс с помощью перегрузки операторов. В главе 15 мы сделаем его более универсальным, преобразовав в шаблон класса, и используем концепты C++20, чтобы он стал еще лучше.

11.6.1. Специальные функции класса

Каждый пользовательский класс может содержать пять специальных функций, которые мы определим в классе `myArray`:

- **копирующий конструктор (copy constructor)**;
- **оператор копирующего присваивания (copy assignment operator)**;
- **перемещающий конструктор (move constructor)**;
- **оператор перемещающего присваивания (move assignment operator)**;
- **деструктор**.

Копирующий конструктор и оператор копирующего присваивания реализуют **семантику копирования (copy semantics)** класса, то есть такой процесс копирования объекта, когда он передается в функцию по значению, возвращается из функции по значению или присваивается другому объекту `myArray`. Перемещающий конструктор и оператор перемещающего присваивания реализуют **семантику перемещения (move semantics)** класса, которая устраняет ненужное копирование объектов, когда в копиях нет необходимости. Мы будем обсуждать детали этих специальных функций классов по мере их реализации в нашем классе.

11.6.2. Работа с классом myArray

Иллюстрации 11.3–11.5 демонстрируют работу класса `myArray` и его богатый набор перегруженных операторов. Код, показанный на илл. 11.3, тестирует различные возможности класса `myArray`. Определение класса показано на илл. 11.4, а определения его функций — на илл. 11.5. Мы разделили код и выходные данные на небольшие фрагменты для удобства обсуждения. Для наглядности многие функции класса, в том числе все его специальные функции, выводят сообщения, показанные в выходных данных.

Функция getArrayByValue

В этой программе мы будем вызывать функцию `getArrayByValue` (илл. 11.3, строки 10–13), чтобы создать локальный объект `myArray` через конструктор класса `myArray`, получающий список инициализаторов. Функция `getArrayByValue` возвращает этот локальный объект по значению.

```
 1 // fig11_03.cpp
 2 // Тестирование класса MyArray
 3 #include <fmt/format.h>
 4 #include <iostream>
 5 #include <stdexcept>
 6 #include <utility> // Для std::move
 7 #include "MyArray.h"
 8
 9 // Функция для возврата MyArray по значению
10 MyArray getArrayByValue() {
```

```
11    MyArray localInts{10, 20, 30}; // Создаем MyArray с тремя элементами
12    return localInts; // Возврат по значению создает rvalue
13 }
14
```

Илл. 11.3 | Тестирование класса MyArray

Создание объектов myArray, отображение их размера и содержимого

В строках 16–17 создаются объекты: `ints1` с семью элементами и `ints2` с десятью элементами. Каждый из них вызывает конструктор класса `myArray`, который получает количество элементов и инициализирует элементы нулевым значением. В строках 20–21 отображается размер объекта `ints1` и выводится его содержимое с помощью перегруженного оператора вставки в поток (`<<`). В строках 24–25 выполняются аналогичные действия с объектом `ints2`.

```
15 int main() {
16    MyArray ints1(7); // MyArray с 7 элементами. Внимание: круглые скобки!
17    MyArray ints2(10); // MyArray с 10 элементами. Внимание: круглые скобки!
18
19    // Выводим размер и содержимое объекта ints1
20    std::cout << fmt::format("\nints1 size: {}\ncontents: ", ints1.size())
21       << ints1; // Используем перегруженный оператор <<
22
23    // Выводим размер и содержимое объекта ints2
24    std::cout << fmt::format("\nints2 size: {}\ncontents: ", ints2.size())
25       << ints2; // Используем перегруженный оператор <<
26
```

```
MyArray(size_t) constructor
MyArray(size_t) constructor

ints1 size: 7
contents: {0, 0, 0, 0, 0, 0, 0}

ints2 size: 10
contents: {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

Круглые скобки вместо фигурных для вызова конструктора

До сих пор мы использовали скобочную инициализацию {} для передачи аргументов конструкторам. В строках 16–17 мы используем круглые скобки () для вызова конструктора `myArray`, получающего размер объекта. Мы делаем это потому, что наш класс `myArray`, подобно классам стандартной библиотеки `array` и `vector`, поддерживает конструирование объекта по списку инициализаторов, содержащему значения элементов. Например, когда компилятор видит инструкцию:

```
MyArray ints1{7};
```

он вызывает конструктор, принимающий список целых чисел, а не конструктор с одним аргументом, получающий размер.

Перегруженный оператор извлечения из потока для заполнения объекта myArray

Далее в строке 28 пользователю предлагается ввести 17 целых чисел. В строке 29 используется перегруженный оператор извлечения из потока (>>), чтобы получить первые семь значений для объекта ints1 и еще десять значений для объекта ints2 (напомним, что каждый объект myArray знает свой размер). В строке 31 отображается обновленное содержимое каждого объекта myArray с помощью перегруженного оператора вставки в поток (<<).

```
27    // Считываем данные для объектов ints1 и ints2, выводим их содержимое
28    std::cout << "\n\nEnter 17 integers: ";
29    std::cin >> ints1 >> ints2; // Используем перегруженный оператор >>
30
31    std::cout << "\nints1: " << ints1 << "\nints2: " << ints2;
32
```

```
Enter 17 integers: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

ints1: {1, 2, 3, 4, 5, 6, 7}
ints2: {8, 9, 10, 11, 12, 13, 14, 15, 16, 17}
```

Перегруженный оператор неравенства (!=)

В строке 36 для проверки работы перегруженного оператора неравенства (!=) вычисляется условие:

```
ints1 != ints2
```

Выходные данные показывают, что объекты myArray не равны. Два объекта будут равны, если они имеют одинаковое количество элементов и соответствующие значения всех элементов идентичны. Как вы скоро увидите, мы определим только перегруженный оператор ==. В C++20 компилятор автоматически генерирует перегруженный оператор !=, если вы предоставили оператор == для вашего типа. Оператор != просто возвращает значение, противоположное оператору ==.

```
33    // Перегруженный оператор неравенства (!=)
34    std::cout << "\n\nEvaluating: ints1 != ints2\n";
35
36    if (ints1 != ints2) {
37       std::cout << "ints1 and ints2 are not equal\n\n";
38    }
39
```

```
Evaluating: ints1 != ints2
ints1 and ints2 are not equal
```

Инициализация нового объекта myArray копией существующего объекта myArray

В строке 41 создается объект ints3 и инициализируется копией объекта ints1. Эта инструкция вызывает копирующий конструктор класса myArray для копирования

элементов `ints1` в `ints3`. Копирующий конструктор вызывается каждый раз, когда требуется копия объекта, например:

- при передаче объекта в функцию по значению;
- при возврате объекта из функции по значению;
- при инициализации объекта копией другого объекта того же класса.

В строках 44–45 отображаются размер и содержимое объекта `ints3`, что подтверждает корректную инициализацию `ints3` копирующим конструктором.

```
40    // Создаем объект ints3 как копию объекта ints1
41    MyArray ints3{ints1}; // Вызываем копирующий конструктор
42
43    // Выводим размер и содержимое объекта ints3
44    std::cout << fmt::format("\nints3 size: {}\ncontents: ", ints3.size())
45       << ints3;
46
```

```
MyArray copy constructor

ints3 size: 7
contents: {1, 2, 3, 4, 5, 6, 7}
```

Когда пользовательский класс, в нашем случае `myArray`, содержит и копирующий, и перемещающий конструкторы, компилятор выбирает нужный конструктор в зависимости от контекста. В строке 41 компилятор выбирает копирующий конструктор класса `myArray`, потому что переменные, такие как `ints1`, являются *lvalue*. Как вы скоро увидите, перемещающий конструктор получает ссылку *rvalue*, являющуюся частью семантики перемещения (C++11). Ссылка *rvalue* может не ссылаться на значение *lvalue*.

Копирующий конструктор также может быть вызван инструкцией:

```
MyArray ints3 = ints1;
```

В определении объекта знак равенства не означает присваивание. Он вызывает копирующий конструктор с одним аргументом, передавая в качестве аргумента значение справа от символа =.

Перегруженный оператор копирующего присваивания (=)

В строке 49 объект `ints2` присваивается объекту `ints1` для тестирования перегруженного оператора копирующего присваивания (=). Традиционные массивы не поддерживают эту операцию. Имя массива не является изменяемым значением *lvalue*, поэтому попытка присвоить имя массива вызывает ошибку компиляции. В строке 51 отображается содержимое двух объектов, что подтверждает их идентичность. Объект `ints1` изначально содержал семь целых чисел, но перегруженный оператор изменяет размер массива, чтобы вместить копию объекта `ints2` из десяти элементов. Как и в случае с копирующими и перемещающими конструкторами, если класс содержит и оператор копирующего присваивания, и оператор переме-

щающего присваивания, компилятор делает выбор между ними, основываясь на аргументах. В этом случае объект `ints2` является переменной и, следовательно, значением *lvalue*, поэтому вызывается оператор копирующего присваивания. Обратите внимание на выходные данные: инструкция в строке 49 вызывает копирующий конструктор и деструктор — вы поймете почему, когда мы рассмотрим реализацию оператора присваивания в разделе 11.6.6.

```
47    // Перегруженный оператор присваивания (=)
48    std::cout << "\n\nAssigning ints2 to ints1:\n";
49    ints1 = ints2; // Внимание: до присваивания ints1 был меньше, чем ints2
50
51    std::cout << "\nints1: " << ints1 << "\nints2: " << ints2;
52
```

```
Assigning ints2 to ints1:
MyArray copy assignment operator
MyArray copy constructor
MyArray destructor

ints1: {8, 9, 10, 11, 12, 13, 14, 15, 16, 17}
ints2: {8, 9, 10, 11, 12, 13, 14, 15, 16, 17}
```

Перегруженный оператор равенства (==)

В строке 56 объекты `inst1` и `ints2` сравниваются перегруженным оператором равенства (==), что подтверждает их идентичность после присваивания в строке 49.

```
53    // Перегруженный оператор равенства (==)
54    std::cout << "\n\nEvaluating: ints1 == ints2\n";
55
56    if (ints1 == ints2) {
57       std::cout << "ints1 and ints2 are equal\n\n";
58    }
59
```

```
Evaluating: ints1 == ints2
ints1 and ints2 are equal
```

Перегруженный оператор индекса ([])

В строке 61 перегруженный оператор индекса (`[]`) используется для доступа к элементу `ints1[5]`, который находится в пределах границ объекта `ints1`. Имя с индексом используется для получения значения, хранящегося в `ints1[5]`. В строке 65 `ints1[5]` используется в левой части присваивания в качестве изменяемого значения *lvalue*[1], чтобы присвоить элементу с индексом 5 объекта `ints1` новое

[1] Напомним, что значение lvalue может быть объявлено как константа. В этом случае оно не будет изменяемым.

значение **1000**. Далее мы увидим, что оператор индекса возвращает ссылку в качестве изменяемого значения *lvalue* после подтверждения допустимости указанного индекса. В строке 71 мы пытаемся присвоить новое значение **1000** несуществующему элементу `ints1[15]`. Этот индекс находится за пределами границ объекта `int1`, поэтому перегруженный оператор индекса выдает исключение `out_of_range`. В строках 73–75 исключение перехватывается и вызывается его функция `what`, выводящая сообщение об ошибке.

```
60    // Используем перегруженный оператор индекса для создания значения rvalue
61    std::cout << fmt::format("ints1[5] is {}\n\n", ints1[5]);
62
63    // Используем перегруженный оператор индекса для создания значения lvalue
64    std::cout << "Assigning 1000 to ints1[5]\n";
65    ints1[5] = 1000;
66    std::cout << "ints1: " << ints1;
67
68    // Пытаемся получить доступ к несуществующему элементу
69    try {
70       std::cout << "\n\nAttempt to assign 1000 to ints1[15]\n";
71       ints1[15] = 1000; // Ошибка: недопустимый индекс!
72    }
73    catch (const std::out_of_range& ex) {
74       std::cout << fmt::format("An exception occurred: {}\n", ex.what());
75    }
76
```

```
ints1[5] is 13

Assigning 1000 to ints1[5]
ints1: {8, 9, 10, 11, 12, 1000, 14, 15, 16, 17}

Attempt to assign 1000 to ints1[15]
An exception occurred: Index out of range
```

Оператор индекса [] нужен не только для работы с массивами. Его также можно использовать, например, для выбора элементов из других контейнеров, поддерживающих коллекции элементов, таких как `string` (коллекции символов) и `map` (коллекции пар «ключ — значение», которые мы рассмотрим в главе 13). Кроме того, когда определены перегруженные функции этого оператора, индексы не обязательно должны быть целыми числами. В главе 13 мы рассмотрим класс `map` стандартной библиотеки, который допускает индексы других типов, например `string`.

Создание и инициализация объекта ints4 функцией getArrayByValue

В строке 80 объект `ints4` инициализируется результатом вызова функции `getArrayByValue` (строки 10–13), которая создает локальный объект класса `myArray` (содержащий три целых числа **10**, **20** и **30**) и возвращает объект по значению. В строках 82–83 отображаются размер и содержимое нового объекта `ints4`.

```
77    // Инициализируем ints4 содержимым объекта MyArray, возвращенного
78    // функцией getArrayByValue; выводим размер и содержимое
79    std::cout << "\nInitialize ints4 with temporary MyArray object\n";
80    MyArray ints4{getArrayByValue()};
81
82    std::cout << fmt::format("\nints4 size: {}\ncontents: ", ints4.size())
83       << ints4;
84
```

```
Initialize ints4 with temporary MyArray object
MyArray(initializer_list) constructor

ints4 size: 3
contents: {10, 20, 30}
```

Оптимизация именованного возвращаемого значения (NRVO)

Напомним из определения функции `getArrayByValue` (строки 10–13), что она создает и инициализирует локальный объект класса `myArray` с помощью конструктора, получающего список инициализаторов с целочисленными значениями (строка 11). Этот конструктор отображает при каждом вызове строку:

```
MyArray(initializer_list) constructor
```

Затем функция `getArrayByValue` возвращает этот локальный объект по значению (строка 12). Вы могли ожидать, что возврат объекта по значению приведет к созданию временной копии для использования в вызывающем коде. Если бы это было так, то вызывался бы копирующий конструктор для копирования локального объекта. Вы также могли ожидать, что локальный объект выйдет из области видимости и его деструктор будет вызван, когда функция `getArrayByValue` вернет управление вызывающему коду. Однако ни копирующий конструктор, ни деструктор не вывели никаких сообщений в выходных данных.

Причина заключается в том, что компилятор повышает быстродействие методом, который называется **оптимизацией именованного возвращаемого значения (named return value optimization**, или сокращенно **NRVO)**. Когда компилятор видит, что локальный объект создан, возвращен из функции по значению, а затем используется для инициализации объекта в точке вызова, компилятор создает объект непосредственно в точке вызова, где он будет использован. При этом не создается временный объект и не происходят дополнительные вызовы конструктора и деструктора, упомянутые выше. Первоначально это была необязательная оптимизация, но в C++17 она стала обязательной[1,2].

---

[1] Сай Бренд (Sy Brand), «Guaranteed Copy Elision Does Not Elide Copies», блог команды C++, 18 февраля 2019 г. Был доступен: 15 сентября 2023 г. https://devblogs.microsoft.com/cppblog/guaranteed-copy-elision-does-not-elide-copies/.

[2] Ричард Смит (Richard Smith), «Guaranteed Copy Elision Through Simplified Value Categories», 27 сентября 2015 г. Был доступен: 15 сентября 2023 г. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0135r0.html.

Создание и инициализация объекта ints5 значением rvalue, возвращенным функцией std::move

Копирующий конструктор вызывается при инициализации одного объекта класса `myArray` другим, который представлен значением *lvalue*. Копирующий конструктор копирует содержимое своего аргумента. Это похоже на операцию «копировать и вставить» в текстовом редакторе: после такой операции у вас есть две копии данных.

C++ также поддерживает семантику перемещения[1], которая помогает компилятору избежать дополнительной нагрузки, вызванной бесполезным копированием объектов. Перемещение работает аналогично операции «вырезать и вставить» в текстовом редакторе: данные перемещаются из одного места в другое. Перемещающий конструктор перемещает в новый объект все ресурсы исходного, уже ненужного объекта. Такой конструктор получает ссылку *rvalue* (C++11), которая, как вы увидите, объявлена как *ИмяТипа*`&&`. Ссылки *rvalue* могут ссылаться только на значения *rvalue*. Как правило, это временные объекты или объекты, подлежащие уничтожению, — *xvalue* (сокращение от «eXpiring value»).

В строке 88 перемещающий конструктор класса `myArray` используется для инициализации объекта `ints5`, а в строках 90–91 отображаются размер и содержимое нового объекта. Объект `ints4` является значением *lvalue*, поэтому он не может быть передан непосредственно в перемещающий конструктор. Если вам больше не нужны ресурсы объекта, вы можете преобразовать его из *lvalue* в *rvalue*, передав этот объект функции `std::move` стандартной библиотеки (C++11, заголовок `<utility>`). Эта функция приводит свой аргумент к ссылке *rvalue*[2], сообщая компилятору, что содержимое объекта `ints4` больше не требуется. Итак, в строке 88 вызывается перемещающий конструктор класса `myArray` и содержимое объекта `ints4` перемещается в объект `ints5`.

```
85    // Преобразуем ints4 в ссылку rvalue функцией std::move
86    // и используем результат для инициализации объекта ints5
87    std::cout << "\n\nInitialize ints5 with result of std::move(ints4)\n";
88    MyArray ints5{std::move(ints4)}; // Вызываем перемещающий конструктор
89
90    std::cout << fmt::format("\nints5 size: {}\ncontents: ", ints5.size())
91       << ints5
92       << fmt::format("\n\nSize of ints4 is now: {}", ints4.size());
93
```

```
Initialize ints5 with result of std::move(ints4)
MyArray move constructor

ints5 size: 3
contents: {10, 20, 30}

Size of ints4 is now: 0
```

[1] Клаус Игльбергер (Klaus Iglberger), «Back to Basics: Move Semantics», 16 июня 2019 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=St0MNEU5b0o.

[2] Точнее говоря, xvalue.

Рекомендуется использовать функцию `std::move` так, как показано в этом коде, только если вы уверены, что переданный в `std::move` объект больше не будет использоваться. После перемещения исходный объект остается **пустым (empty)**, поэтому с ним можно выполнить только две допустимые операции:

- уничтожить его;
- использовать его в левой части выражения присваивания, чтобы присвоить ему новое значение.

Как правило, вызывать функции пустого объекта недопустимо. Мы сделали это в строке 92 только для того, чтобы убедиться, что перемещающий конструктор действительно переместил ресурсы объекта: в выходных данных показан размер объекта `ints4`, равный `0`.

Присваивание объекта ints5 объекту ints4 оператором перемещающего присваивания

В строке 96 оператор перемещающего присваивания класса `myArray` используется для перемещения содержимого объекта `ints5` (`10`, `20` и `30`) обратно в `ints4`. В строках 98–99 отображаются размер и содержимое обновленного объекта `ints4`. В строке 96 значение *lvalue* `ints5` явно преобразуется в ссылку *rvalue* функцией `std::move`. Это означает, что объект `ints5` больше не нуждается в своих ресурсах, поэтому компилятор может переместить их в `ints4` и вызывает оператор перемещающего присваивания. Только в демонстрационных целях мы выводим в строке 100 размер объекта `ints5`, показывая, что оператор перемещающего присваивания действительно переместил ресурсы. Напомним, что вы никогда не должны вызывать функции пустого объекта!

```
94    // Перемещаем содержимое объекта ints5 в объект ints4
95    std::cout << "\n\nMove ints5 into ints4 via move assignment\n";
96    ints4 = std::move(ints5); // Перемещающее присваивание
97
98    std::cout << fmt::format("\nints4 size: {}\ncontents: ", ints4.size())
99       << ints4
100       << fmt::format("\n\nSize of ints5 is now: {}", ints5.size());
101
```

```
Move ints5 into ints4 via move assignment
MyArray move assignment operator

ints4 size: 3
contents: {10, 20, 30}

Size of ints5 is now: 0
```

Преобразование типа myArray в bool для проверки наличия элементов в ints5

Многие языки программирования позволяют использовать объект контейнерного класса в качестве условия, показывающего наличие элементов в контейнере. Для класса `myArray` мы определили конвертирующий оператор `bool`, который возвра-

щает `true`, если объект `myArray` содержит элементы (то есть его размер больше `0`), или `false` при отсутствии элементов. В тех случаях, когда нужны значения `bool`, например, в условиях управляющих инструкций, C++ может неявно вызывать этот конвертирующий оператор. Это называется **контекстным преобразованием (contextual conversion)**. В строке 103 объект `ints5` используется в качестве условия, которое вызывает конвертирующий оператор `bool`. Мы только что переместили ресурсы объекта `ints5` в `ints4`, и оператор возвращает `false`, поскольку `ints5` теперь пуст. Еще раз подчеркнем, что вы никогда не должны вызывать функции пустых объектов: мы делаем это только для проверки корректности перемещения ресурсов из объекта `ints5`.

```
102    // Проверяем контекстным преобразованием в bool наличие элементов в ints5
103    if (ints5) {
104       std::cout << "\n\nints5 contains elements\n";
105    }
106    else {
107       std::cout << "\n\nints5 is empty\n";
108    }
109
```

```
ints5 is empty
```

Преинкремент элементов ints4 перегруженным оператором префиксного инкремента (++)

Некоторые библиотеки поддерживают **массовые (broadcast)** операции, то есть могут применять одну и ту же операцию к каждому элементу структуры данных. Например, рассмотрим популярную библиотеку `NumPy` языка программирования Python. Структура `ndarray` (n-мерный массив) этой библиотеки перегружает многие арифметические операторы. Они выполняют математические операции над каждым элементом объекта типа `ndarray`. В `NumPy` следующий код на языке Python прибавляет `1` к каждому элементу объекта `numbers` типа `ndarray` — инструкция итерации не требуется:

```
numbers += 1 # В языке Python нет оператора ++
```

Мы добавили эту возможность в класс `myArray`. В строке 111 отображается содержимое объекта `ints4`, затем в строке 112 выполняется префиксный инкремент всего объекта (выражение `++ints4`), то есть к каждому его элементу прибавляется 1. Результатом этого выражения является обновленный объект. Мы выводим его содержимое перегруженным оператором вставки в поток (`<<`).

```
110    // Прибавляем 1 к каждому элементу ints4 перегруженным оператором
          префиксного инкремента
111    std::cout << "\nints4: " << ints4;
112    std::cout << "\npreincrementing ints4: " << ++ints4;
113
```

```
ints4: {10, 20, 30}
preincrementing ints4: {11, 21, 31}
```

Постинкремент элементов ints4 перегруженным оператором постфиксного инкремента (++)

В строке 115 выражение `ints4++` увеличивает все элементы объекта с помощью оператора постфиксного инкремента. Напомним, что эта операция возвращает предыдущее значение своего операнда, что подтверждается выводом программы. Выходные данные, показывающие моменты вызовов копирующего конструктора и деструктора, взяты из реализации оператора постфиксного инкремента, обсуждаемой в разделе 11.6.14.

```
114    // Прибавляем 1 к каждому элементу ints4 перегруженным оператором
          постфиксного инкремента
115    std::cout << "\n\npostincrementing ints4: " << ints4++ << "\n";
116    std::cout << "\nints4 now contains: " << ints4;
117
```

```
postincrementing ints4: MyArray copy constructor
{11, 21, 31}
MyArray destructor

ints4 now contains: {12, 22, 32}
```

Прибавление значения к каждому элементу объекта ints4 перегруженным оператором сложения с присваиванием (+=)

Для массовых операций класс `myArray` также предоставляет перегруженный составной оператор присваивания (`+=`), прибавляющий целое число к каждому элементу. В строке 119 мы суммируем каждый элемент объекта `ints4` с числом 7 и отображаем обновленное содержимое. Обратите внимание, что неперегруженные версии операторов `++` и `+=` по-прежнему работают с отдельными элементами `myArray`, которые являются целыми числами.

```
118    // Прибавляем число к каждому элементу ints4 перегруженным оператором
          постфиксного инкремента
119    std::cout << "\n\nAdd 7 to every ints4 element: " << (ints4 += 7)
120       << "\n";
121    }
```

```
Add 7 to every ints4 element: {19, 29, 39}
```

Уничтожение всех объектов класса myArray

Когда функция `main` завершается, вызываются деструкторы пяти объектов, созданных в функции `main`. Они выводят последние пять строк выходных данных программы.

```
MyArray destructor
MyArray destructor
MyArray destructor
MyArray destructor
MyArray destructor
```

11.6.3. Определение класса myArray

Рассмотрим заголовок класса `myArray` (илл. 11.4). Упоминая каждую функцию класса в заголовке, мы затем рассматриваем реализацию этой функции в коде на илл. 11.5. Для удобства обсуждения мы разделили файл реализации функций класса на фрагменты.

На илл. 11.4 в строках 53–54 объявляются скрытые элементы данных класса `myArray`:

- `m_size` хранит количество элементов.
- `m_ptr` — это умный указатель `unique_ptr`, который управляет динамически размещаемым массивом, содержащим элементы объекта `myArray`. Когда объект `myArray` выходит из области видимости, его деструктор вызывает деструктор умного указателя `unique_ptr`, автоматически освобождая выделенную память.

Во всех реализациях функций этого класса мы используем некоторые возможности декларативного программирования в функциональном стиле C++, рассмотренные в главах 6 и 7. Мы также представляем вам три алгоритма стандартной библиотеки: `copy`, `for_each` и `equal`.

```
 1 // Fig. 11.4: MyArray.h
 2 // Определение класса MyArray с перегруженными операторами
 3 #pragma once
 4 #include <initializer_list>
 5 #include <iostream>
 6 #include <memory>
 7
 8  class MyArray final {
 9    // Перегруженный оператор извлечения из потока
10    friend std::istream& operator>>(std::istream& in, MyArray& a);
11
12    // Используется оператором копирующего присваивания для реализации идиомы
        копирования и обмена
13    friend void swap(MyArray& a, MyArray& b) noexcept;
14
15 public:
16    explicit MyArray(size_t size); // Создаем MyArray, содержащий size элементов
17
18    // Создаем MyArray со скобочной инициализацией списком целых чисел
19    explicit MyArray(std::initializer_list<int> list);
20
21    MyArray(const MyArray& original); // Копирующий конструктор
22    MyArray& operator=(const MyArray& right); // Оператор копирующего
                                                   присваивания
23
24    MyArray(MyArray&& original) noexcept; // Перемещающий конструктор
25    MyArray& operator=(MyArray&& right) noexcept; // Оператор перемещающего
                                                      присваивания
26
```

Окончание ⇨

```
27     ~MyArray(); // Деструктор
28
29     size_t size() const noexcept {return m_size;}; // Возвращаем размер
30     std::string toString() const; // Создаем строковое представление
31
32     // Оператор равенства
33     bool operator==(const MyArray& right) const noexcept;
34
35     // Оператор индекса для неконстантных объектов, возвращающий изменяемое
          значение lvalue
36     int& operator[](size_t index);
37
38     // Оператор индекса для константных объектов, возвращающий неизменяемое
          значение lvalue
39     const int& operator[](size_t index) const;
40
41     // Преобразуем MyArray в значение bool (true - есть элементы, false - нет)
42     explicit operator bool() const noexcept {return size() != 0;}
43
44     // Преинкрементируем каждый элемент и возвращаем обновленный MyArray
45     MyArray& operator++();
46
47     // Постинкрементируем каждый элемент и возвращаем копию исходного MyArray
48     MyArray operator++(int);
49
50     // Прибавляем число к каждому элементу и возвращаем обновленный MyArray
51     MyArray& operator+=(int value);
52 private:
53     size_t m_size{0}; // Размер традиционного массива на основе указателя
54     std::unique_ptr<int[]> m_ptr; // Умный указатель на массив целых чисел
55 };
56
57 // Перегруженный оператор << не дружественный, без доступа к скрытым данным
58 std::ostream& operator<<(std::ostream& out, const MyArray& a);
```

Илл. 11.4 | Определение класса MyArray с перегруженными операторами

11.6.4. Конструктор, задающий размер объекта myArray

В строке 16 на илл. 11.4:

```
explicit MyArray(size_t size); // Создаем MyArray, содержащий size элементов
```

объявляется конструктор, задающий количество элементов объекта `myArray`. Определение конструктора (илл. 11.5, строки 15–19) выполняет несколько задач:

- В строке 16 элемент данных `m_size` инициализируется аргументом `size`.
- В строке 17 элемент данных `m_ptr` инициализируется указателем `unique_ptr` через шаблон функции стандартной библиотеки `make_unique` (см. раздел 11.5). Здесь мы используем его, чтобы выделить динамическую па-

мять для массива целых чисел, количество которых равно `size`. Функция `make_unique` инициализирует выделенную память, присваивая элементам массива значение `0`.

- В демонстрационных целях в строке 18 выводится сообщение о вызове конструктора. Мы делаем это во всех специальных функциях и конструкторах класса `myArray`, чтобы дать визуальное подтверждение их вызовам.

```
 1 // Fig. 11.5: MyArray.cpp
 2 // Определения функций класса MyArray и дружественных функций
 3 #include <algorithm>
 4 #include <fmt/format.h>
 5 #include <initializer_list>
 6 #include <iostream>
 7 #include <memory>
 8 #include <span>
 9 #include <sstream>
10 #include <stdexcept>
11 #include <utility>
12 #include "MyArray.h" // Определение класса MyArray
13
14 // Конструктор для создания MyArray, в котором size элементов со значением 0
15 MyArray::MyArray(size_t size)
16    : m_size{size},
17      m_ptr{std::make_unique<int[]>(size)} {
18    std::cout << "MyArray(size_t) constructor\n";
19 }
20
```

Илл. 11.5 | Определения функций класса MyArray и дружественных функций

11.6.5. Скобочная инициализация в конструкторе (C++11)

На илл. 6.2 мы инициализировали объект `std::array` списком инициализаторов:

```
std::array<int, 5> n{32, 27, 64, 18, 95};
```

Вы можете использовать скобочную инициализацию и для объектов пользовательских классов, предоставив конструктору параметр `std::initializer_list`, как это сделано в строке 19 на илл. 11.4:

```
explicit MyArray(std::initializer_list<int> list);
```

Шаблон класса `std::initializer_list` определен в заголовке `<initializer_list>`. С помощью этого конструктора мы можем создавать объекты `myArray`, инициализирующие свои элементы через конструктор:

```
MyArray ints{10, 20, 30};
```

или:

```
MyArray ints = {10, 20, 30};
```

Каждая из этих инструкций создает объект `myArray` из трех элементов: `10`, `20` и `30`. В классе, предоставляющем конструктор с объектом `initializer_list`, другие конструкторы с одним аргументом должны вызываться с круглыми, а не фигурными скобками. Конструктор со скобочной инициализацией (строки 22–29) имеет один параметр `initializer_list<int>` с именем `list`. Вы можете определить количество элементов, вызвав функцию `size` (строка 23).

```
21 // Конструктор myArray, принимающий список инициализаторов
22 MyArray::MyArray(std::initializer_list<int> list)
23    : m_size{list.size()}, m_ptr{std::make_unique<int[]>(list.size())} {
24    std::cout << "MyArray(initializer_list) constructor\n";
25
26    // Копируем элементы из списка в массив int, которым управляет m_ptr
27    // Функция m_ptr.get() возвращает первый адрес памяти этого массива
28    std::copy(std::begin(list), std::end(list), m_ptr.get());
29 }
30
```

Для списочной инициализации нового объекта `myArray` в строке 28 используется алгоритм `copy` стандартной библиотеки (заголовок `<algorithm>`). Он копирует все элементы из `initializer_list` в новый объект `myArray`. Алгоритм `copy` копирует каждый элемент в диапазоне, указанном его первыми двумя аргументами: началом и концом объекта `initializer_list`. Третий аргумент алгоритма `copy` указывает место для копирования элементов. Функция `get` умного указателя `unique_ptr` возвращает значение `int*`, которое указывает на первый элемент целочисленного массива, лежащего в основе объекта `myArray`.

11.6.6. Копирующий конструктор и оператор копирующего присваивания

В разделах 9.15 и 9.17 мы представили вам оператор копирующего присваивания по умолчанию и копирующий конструктор по умолчанию, сгенерированные компилятором. Они выполняли операции копирования элементов. В C++ Core Guidelines рекомендуется разрабатывать классы с автоматически сгенерированными копирующими конструкторами, операторами копирующего присваивания, перемещающими конструкторами, операторами перемещающего присваивания и деструкторами. Это называется **«правилом нуля» (Rule of Zero)**[1]. Вы можете достичь этого, составляя элементы данных каждого класса из основных типов и объектов таких классов, которые не требуют от вас реализации пользовательской обработки ресурсов. Например, это классы стандартной библиотеки `array` и `vector`, использующие RAII для управления ресурсами.

[1] C++ Core Guidelines, «C.20: If You Can Avoid Defining Any Default Operations, Do». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-zero.

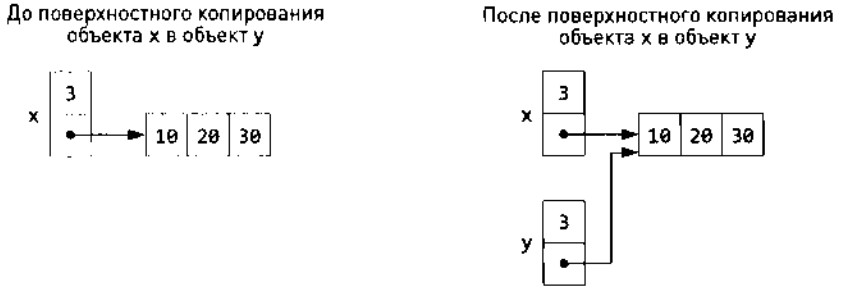
«Правило пяти»

Специальные функции по умолчанию хорошо работают со значениями основных типов, например `int` или `double`. Но как быть с объектами таких типов, которые управляют собственными ресурсами (например, указателями на динамически выделяемую память)? В классе, управляющем собственными ресурсами, должны быть определены все пять специальных функций. Согласно рекомендациям C++ Core Guidelines, если в классе определена хотя бы одна из специальных функций, необходимо определить и все остальные[1], как сделано в нашем примере. Это называется **«правилом пяти» (Rule of Five)**.

Даже для классов со специальными функциями, сгенерированными компилятором, некоторые эксперты рекомендуют объявлять их явно в определении класса с помощью обозначения `= default` (мы рассматривали его в главе 10). Это называется «Rule of Five defaults»[2]. Вы также можете явно удалить созданные компилятором специальные функции, чтобы исключить соответствующую функциональность, добавив к их прототипам обозначение `= delete` (C++11). В классе `unique_ptr` это реализовано для копирующего конструктора и для оператора копирующего присваивания.

Поверхностное копирование

Копирующий конструктор и оператор копирующего присваивания, сгенерированные компилятором, выполняют только **поверхностное (shallow)** копирование элементов данных. Если элемент данных является указателем на динамически выделяемую память, то при копировании этого элемента будет скопирован только адрес памяти. На следующей диаграмме изображен объект `x`, содержащий указатель на динамически размещаемый массив и количество элементов в этом массиве (`3`). Для примера предположим, что элемент указателя — это просто `int*`, а не `unique_ptr`.



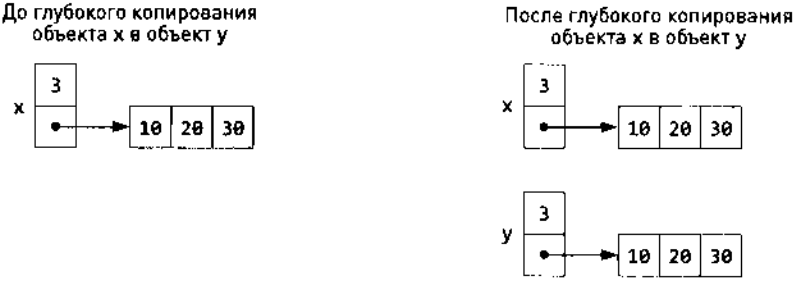
[1] C++ Core Guidelines, «C.21: If You Define or =delete Any Copy, Move, or Destructor Function, Define or =delete Them All». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-five.

[2] Скотт Майерс (Scott Meyers), «A Concern About the Rule of Zero», 13 марта 2014 г. Был доступен: 15 сентября 2023 г. http://scottmeyers.blogspot.com/2014/03/a-concern-about-rule-of-zero.html.

Предположим, что мы хотим скопировать x в новый объект y. Если копирующий конструктор просто скопирует указатель из x в y, оба указателя будут направлены на одну и ту же область динамически выделяемой памяти, как показано в правой части диаграммы. Первый выполняемый деструктор освободит эту область памяти, которая является общей для двух объектов, и тогда второй указатель будет указывать на память, которая не выделена. Эта ситуация называется **висячим указателем (dangling pointer)**. Если программа попытается разыменовать висячий указатель, произойдет ошибка (например, аварийное завершение программы), поскольку обращение к освобожденной области памяти приводит к неопределенному поведению.

Глубокое копирование

Когда класс управляет собственными ресурсами, копирование надо выполнять осторожно, чтобы избежать ловушек поверхностного копирования. Класс, управляющий ресурсами своих объектов, должен определить собственный копирующий конструктор и собственный перегруженный оператор копирующего присваивания, чтобы выполнять **глубокое (deep)** копирование. На следующей диаграмме показано глубокое копирование объекта x в объект y. После глубокого копирования каждый из объектов имеет в динамически выделяемой памяти собственную копию массива, содержащего числа `10`, `20` и `30`. Отсутствие копирующего конструктора и перегруженного оператора копирующего присваивания для объектов класса, содержащего указатели на динамически выделяемую память, — это потенциальная логическая ошибка.



Реализация копирующего конструктора

В строке 21 на илл. 11.4

```
MyArray(const MyArray& original); // Копирующий конструктор
```

объявляется копирующий конструктор (он определен в строках 32–41). Его аргумент должен быть константной ссылкой, чтобы конструктор не мог изменять данные соответствующего объекта.

```
31 // Копирующий конструктор; он должен получить ссылку на MyArray
32 MyArray::MyArray(const MyArray& original)
33    : m_size{original.size()},
```

```
34      m_ptr{std::make_unique<int[]>(original.size())} {
35    std::cout << "MyArray copy constructor\n";
36
37    // Копируем элементы из original в массив int, которым управляет m_ptr
38    const std::span<const int> source{
39       original.m_ptr.get(), original.size()};
40    std::copy(std::begin(source), std::end(source), m_ptr.get());
41 }
42
```

Когда копирующий конструктор вызывается для инициализации нового объекта класса `myArray` путем копирования существующего объекта, этот конструктор выполняет следующие задачи:

- В строке 33 элемент `m_size` инициализируется значением, возвращаемым функцией `size` объекта `original`.
- В строке 34 элемент `m_ptr` инициализируется умным указателем `unique_ptr`, возвращенным стандартной библиотечной функцией `make_unique`, которая создает в динамически выделяемой памяти целочисленный массив, содержащий `original.size()` элементов.
- В строке 35 выводится сообщение о вызове конструктора.
- Напомним, что объект `span` — это представление (`view`) непрерывной коллекции элементов, например массива. В строках 38–39 создается объект `span` с именем `source`, представляющий целочисленный массив в динамической памяти, из которого будут копироваться элементы.
- В строке 40 элементы диапазона, представленного началом и концом объекта `source`, копируются в целочисленный массив, лежащий в основе нового объекта класса `myArray`.

Оператор копирующего присваивания (=)

В строке 22 на илл. 11.4:

```
MyArray& operator=(const MyArray& right); // Оператор копирующего присваивания
```

объявляется перегруженный оператор копирующего присваивания (=)[1]. Определение этой функции (строки 44–49) позволяет присвоить один объект `myArray` другому, копируя содержимое из правого операнда в левый. Когда компилятор видит инструкцию:

```
ints1 = ints2;
```

он вызывает функцию `operator=` следующим образом:

```
ints1.operator=(ints2)
```

[1] Этот оператор копирующего присваивания гарантирует, что при возникновении исключения объект класса myArray не будет изменен и не произойдет утечка памяти. Это называется строгой гарантией безопасности исключений (см. раздел 12.3).

```
43 // Оператор копирующего присваивания; реализует идиому копирования и обмена
44 MyArray& MyArray::operator=(const MyArray& right) {
45    std::cout << "MyArray copy assignment operator\n";
46    MyArray temp{right}; // Вызываем копирующий конструктор
47    swap(*this, temp); // Обмениваемся содержимым между данным объектом и temp
48    return *this;
49 }
50
```

Мы можем реализовать перегруженный оператор копирующего присваивания аналогично копирующему конструктору. Однако есть элегантная возможность использовать копирующий конструктор для реализации перегруженного оператора копирующего присваивания: **идиома копирования и обмена (copy-and-swap idiom)**[1,2]. Она заключается в следующем:

- Сначала аргумент копируется в локальный объект `temp` с помощью копирующего конструктора (строка 46). Если при этом не удается выделить память для массива `temp`, возникает исключение `bad_alloc`. В этом случае перегруженный оператор копирующего присваивания прерывает работу без изменения объекта в левой части выражения присваивания.
- В строке 47 используется дружественная функция `swap` (определена в строках 169–172) для обмена содержимым между объектом `*this` (в левой части выражения присваивания) и объектом `temp`.
- Функция возвращает ссылку на текущий объект (`*this` в строке 48), позволяя выполнять каскадные присваивания объектов `MyArray`, такие как `x = y = z`.

Когда функция возвращает управление в точку вызова, вызывается деструктор объекта `temp`, чтобы освободить память, на которую ссылается умный указатель `unique_ptr`. Вызов копирующего конструктора в строке 46 и вызов деструктора в тот момент, когда объект `temp` выходит за пределы области видимости, — это причины двух дополнительных строк вывода, которые вы видели при присваивании объекта `ints2` объекту `ints1` в строке 49 на илл. 11.3.

11.6.7. Перемещающий конструктор и оператор перемещающего присваивания

Во многих случаях операция копирования объекта должна завершаться уничтожением исходного объекта. Например, это относится к локальному объекту, возвращенному из функции по значению. Более эффективно переместить содержимое копируемого объекта в новый объект, чтобы исключить влияние копирования на быстродействие. Это задача перемещающего конструктора и оператора перемещающего присваивания, которые объявлены в строках 24–25 на илл. 11.4:

---

[1] Херб Саттер (Herb Sutter), «Exception-Safe Class Design, Part 1: Copy Assignment». Был доступен: 15 сентября 2023 г. http://www.gotw.ca/gotw/059.htm.

[2] Джек Лилхаммерс (Jack Lilhammers), ответ на «What is the copy-and-Swap Idiom?», 24 апреля 2021 г. Был доступен: 15 сентября 2023 г. https://stackoverflow.com/a/3279550.

```
MyArray(MyArray&& original) noexcept; // Перемещающий конструктор
MyArray& operator=(MyArray&& right) noexcept; // Оператор перемещающего
                                              присваивания
```

Каждый из них получает ссылку *rvalue*, которая обозначается двойным символом амперсанда (`&&`) в отличие от ссылки *lvalue* (`&`). Ссылки *rvalue* помогают реализовать семантику перемещения. Вместо копирования аргумента перемещающий конструктор и оператор перемещающего присваивания перемещают данные своего аргумента, оставляя исходный объект в состоянии, готовом для уничтожения.

Спецификатор noexcept

Начиная с C++11, если функция не генерирует исключения и не вызывает никакие функции, генерирующие исключения, вы должны явно указать, что эта функция не генерирует исключения[1]. Для этого добавьте спецификатор `noexcept` после сигнатуры функции в прототипе и в определении. Для функции класса, объявленной как константа, ключевое слово `noexcept` должно находиться после `const`. Если `noexcept`-функция вызывает другую функцию, выдающую исключение, но не обрабатывает это исключение, программа немедленно завершается.

Спецификация `noexcept` может дополнительно сопровождаться круглыми скобками, содержащими выражение `bool`, которое вычисляется как `true` или `false`. Спецификатор `noexcept` сам по себе эквивалентен записи `noexcept(true)`. Сигнатура функции с `noexcept(false)` указывает, что разработчик класса допускает возможность исключения при вызове функции. В таких случаях программисты клиентского кода могут решить, следует ли помещать вызов этой функции в инструкцию `try`.

Перемещающий конструктор класса myArray

Перемещающий конструктор (строки 52–56) объявляет свой параметр как ссылку *rvalue* (`&&`), указывая, что его аргумент `myArray` должен быть временным объектом. Список инициализаторов элементов перемещает элементы `m_size` и `m_ptr` из аргумента в создаваемый объект.

```
51 // Перемещающий конструктор; он должен получить ссылку rvalue на MyArray
52 MyArray::MyArray(MyArray&& original) noexcept
53    : m_size{std::exchange(original.m_size, 0)},
54      m_ptr{std::move(original.m_ptr)} { // Перемещаем original.m_ptr в m_ptr
55    std::cout << "MyArray move constructor\n";
56 }
57
```

Напомним из раздела 11.6.2, что после перемещения объекта вы можете выполнить только две операции с исходным, уже пустым объектом: уничтожить его или при-

[1] C++ Core Guidelines, «F.6: If Your Function May Not Throw, Declare It noexcept». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-noexcept.

своить ему другой объект. При перемещении ресурсов из объекта он должен быть оставлен в состоянии, позволяющем корректно его уничтожить. Кроме того, он не должен ссылаться на ресурсы, перемещенные в новый объект[1]. Чтобы выполнить эти условия для элемента `m_size`, в строке 53:

- вызывается функция стандартной библиотеки `exchange` (заголовок `<utility>`), устанавливающая для первого аргумента (`original.m_size`) значение второго аргумента (`0`) и возвращающая исходное значение первого аргумента;
- элемент данных `m_size` нового объекта инициализируется значением, возвращаемым функцией `exchange`.

Когда умный указатель `unique_ptr` создается функцией `move`, как в строке 54, его перемещающий конструктор передает владение областью динамической памяти исходного умного указателя `unique_ptr` новому умному указателю `unique_ptr` и устанавливает значение исходного указателя равным `nullptr`[2]. Если ваш класс работает с простыми указателями, вам надо явно установить значение исходного указателя равным `nullptr` или использовать функцию `exchange`, как в строке 53.

Перемещение ничего не перемещает

Хотя мы сказали, что перемещающий конструктор «перемещает элементы» `m_size` и `m_ptr` из объекта-аргумента в конструируемый объект, на самом деле никакого перемещения не происходит[3]:

- Для каждого из основных типов, таких как `size_t` (целое число без знака), значение копируется из элемента исходного объекта в элемент нового объекта.
- Для простого указателя адрес, сохраненный в указателе исходного объекта, копируется в указатель нового объекта.
- Для объекта вызывается его перемещающий конструктор. Перемещающий конструктор умного указателя `unique_ptr` передает владение соответствующей областью динамической памяти, копируя ее адрес из внутреннего простого указателя исходного объекта `unique_ptr` во внутренний простой указатель нового объекта `unique_ptr`. Затем этот конструктор присваивает `nullptr` исходному объекту `unique_ptr`. Это означает, что исходный объект больше не управляет никакими данными.

Следующая диаграмма показывает принцип перемещения исходного объекта `x`, содержащего простой указатель, в новый объект `y`. Обратите внимание, что после

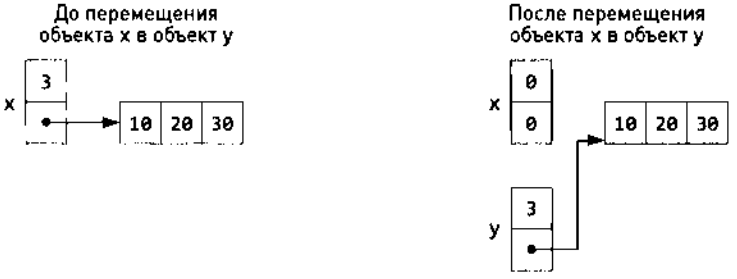
---

[1] C++ Core Guidelines, «C.64: A Move Operation Should Move and Leave Its Source in a Valid State». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-move-semantic.

[2] «std::unique_ptr<T,Deleter>::unique_ptr». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/memory/unique_ptr/unique_ptr.

[3] Тофер Уинворд (Topher Winward), «C++ Moves For People Who Don't Know or Care What Rvalues Are», 17 января 2019 г. Был доступен: 15 сентября 2023 г. https://medium.com/@winwardo/c-moves-for-peoplewho-dont-know-or-care-what-rvalues-are-%EF%B8%8F-56ee122dda7.

перемещения оба элемента объекта x равны 0. Числом 0 в указателе обозначается нулевой указатель (nullptr).



Оператор перемещающего присваивания (=) класса myArray

Оператор перемещающего присваивания (=) (строки 59–69) определяет параметр ссылкой *rvalue* (&&), указывающей, что ресурсы аргумента должны быть перемещены, а не скопированы. В строке 62 проверяется самоприсваивание, в котором объект присваивается самому себе[1]. Когда значение this равно адресу правого операнда (один и тот же объект находится с обеих сторон присваивания), нет необходимости что-либо перемещать.

```
58 // Оператор перемещающего присваивания
59 MyArray& MyArray::operator=(MyArray&& right) noexcept {
60    std::cout << "MyArray move assignment operator\n";
61
62    if (this != &right) { // Отфильтровываем самоприсваивание
63       // Перемещаем данные из right в MyArray
64       m_size = std::exchange(right.m_size, 0); // Обнуляем элемент данных
65       m_ptr = std::move(right.m_ptr);
66    }
67
68    return *this; // Для каскадного присваивания, например: x = y = z
69 }
70
```

Если нет самоприсваивания, то в строках 64–65:

- значение элемента right.m_size перемещается в элемент m_size нового объекта через вызов функции exchange. Эта функция обнуляет значение right.m_size и возвращает исходное значение, которое мы используем для элемента m_size конечного объекта;
- значение элемента right.m_ptr перемещается в m_ptr нового объекта.

[1] «C.65: Make Move Assignment Safe for Self-Assignment». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-move-self.

Как и в перемещающем конструкторе, когда умный указатель `unique_ptr` перемещается через функцию `move`, владение его областью динамической памяти переходит к новому умному указателю `unique_ptr`, а оператор перемещающего присваивания присваивает исходному указателю `unique_ptr` значение `nullptr`. Независимо от того, является ли это самоприсваиванием, функция возвращает текущий объект (`*this`), что позволяет выполнять каскадные присваивания объектов `MyArray`, такие как `x = y = z`.

Операции перемещения должны иметь спецификатор noexcept

Перемещающие конструкторы и операторы перемещающего присваивания никогда не должны выдавать исключения. Они не создают новые ресурсы, а только перемещают существующие. По этой причине в C++ Core Guidelines рекомендуется добавлять к перемещающим конструкторам и операторам перемещающего присваивания спецификатор `noexcept`[1]. Это требование позволяет использовать средства перемещения вашего класса с контейнерами стандартной библиотеки, такими как `vector`.

11.6.8. Деструктор

В строке 27 на илл. 11.4:

```
~MyArray(); // Деструктор
```

объявляется деструктор класса (определенный в строках 73–75), который вызывается, когда объект `myArray` выходит за пределы области видимости. Это автоматически вызовет деструктор указателя `m_ptr`, освобождающий область памяти с целочисленным массивом, динамически выделенную при создании объекта `myArray`. В C++ Core Guidelines отмечается, что деструкторы, выдающие исключения, — это признак плохого программирования. По этой причине рекомендуется добавлять к деструкторам спецификатор `noexcept`[2]. Но если ваш класс не является производным от базового класса с деструктором, объявленным как `noexcept(false)`, компилятор неявно добавляет к деструктору спецификатор `noexcept` по умолчанию.

```
71 // Деструктор мог быть сгенерирован компилятором, но мы добавили его,
72 // чтобы выводить сообщение при уничтожении каждого объекта MyArray
73 MyArray::~MyArray() {
74    std::cout << "MyArray destructor\n";
75 }
76
```

[1] C++ Core Guidelines, «C.66: Make Move Operations noexcept». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-move-noexcept.

[2] C++ Core Guidelines, «C.37: Make Destructors noexcept». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-dtor-noexcept.

11.6.9. Функции toString и size

В строке 29 на илл. 11.4:

```
size_t size() const noexcept {return m_size;}; // Возвращаем размер
```

определяется функция `size`, возвращающая количество элементов объекта класса `myArray`.

В строке 30 на илл. 11.4:

```
std::string toString() const; // Создаем строковое представление
```

объявляется функция `toString` (она определена в строках 78–91), возвращающая строковое представление содержимого объекта. Функция `toString` использует `ostringstream` (см. раздел 8.17) для создания строки типа `string` со значениями элементов, заключенными в фигурные скобки ({}) и разделенными запятой с пробелом.

```
77 // Возвращаем строковое представление объекта MyArray
78 std::string MyArray::toString() const {
79    const std::span<const int> items{m_ptr.get(), m_size};
80    std::ostringstream output;
81    output << "{";
82
83    // Выводим каждый элемент динамического массива в ostringstream
84    for (size_t count{0}; const auto& item : items) {
85       ++count;
86       output << item << (count < m_size ? ", " : "");
87    }
88
89    output << "}";
90    return output.str();
91 }
92
```

11.6.10. Перегрузка операторов равенства и неравенства

В строке 33 на илл. 11.4:

```
bool operator==(const MyArray& right) const noexcept;
```

объявляется перегруженный оператор равенства (==). Операторы сравнения не должны генерировать исключения, поэтому их нужно объявлять со спецификатором `noexcept`[1].

Когда компилятор видит выражение типа `ints1 == ints2`, он вызывает этот перегруженный оператор:

```
ints1.operator==(ints2)
```

[1] C++ Core Guidelines, «C.86: Make == Symmetric with Respect Of Operand Types and noexcept». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-eq.

Функция `operator==` (определенная в строках 95–101) работает следующим образом:

- В строке 97 создается объект `span` с именем `lhs`, представляющий целочисленный массив в левом операнде (`ints1`).
- В строке 98 создается объект `span` с именем `rhs`, представляющий целочисленный массив в правом операнде (`ints2`).
- В строках 99–100 вызывается алгоритм `equal` стандартной библиотеки (заголовок `<algorithm>`) для сравнения соответствующих элементов двух объектов `span`. Первые два аргумента определяют диапазон элементов объекта `lhs`. Последние два аргумента определяют диапазон элементов объекта `rhs`. Если объекты `lhs` и `rhs` имеют разную длину или если какая-либо пара соответствующих элементов отличается, функция `equal` возвращает значение `false`. Если все пары элементов идентичны, функция `equal` возвращает `true`.

```
93 // Проверяем, идентично ли содержимое двух объектов MyArray;
94 // если да - возвращаем true, если нет - возвращаем false
95 bool MyArray::operator==(const MyArray& right) const noexcept {
96    // Сравниваем соответствующие элементы двух объектов MyArray
97    const std::span<const int> lhs{m_ptr.get(), size()};
98    const std::span<const int> rhs{right.m_ptr.get(), right.size()};
99    return std::equal(std::begin(lhs), std::end(lhs),
100       std::begin(rhs), std::end(rhs));
101 }
102
```

Компилятор генерирует оператор !=

В C++20 компилятор автоматически генерирует операторную функцию для оператора `!=`, если вы предоставили оператор `==` для пользовательского типа. До C++20, если для какого-либо класса требовался перегруженный оператор неравенства (`!=`), программисту обычно приходилось определять его, вызывая операторную функцию `==` и возвращая противоположный результат.

Определение операторов сравнения как глобальных функций

В каждом нашем классе, включая класс `myArray`, мы объявляем одноаргументные конструкторы со спецификатором `explicit` для предотвращения неявных преобразований в соответствии с разделом C++ Core Guideline «C.164: Avoid Implicit Conversion Operators»[1]. Вы также можете обратиться к разделу C++ Core Guideline «C.86: Make == Symmetric with Respect to Operand Types and noexcept»[2], где рекомендуется реализовывать операторы сравнения как функции, не являющиеся

[1] C++ Core Guidelines, «C.164: Avoid Implicit Conversion Operators». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ro-conversion.

[2] C++ Core Guidelines, «C.86: Make == Symmetric with Respect to Operand Types and noexcept». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-eq.

элементами классов, если ваш класс поддерживает неявное преобразование объектов других типов в тип вашего класса или наоборот. Это правило применимо ко всем операторам сравнения, но здесь мы рассмотрим оператор ==, поскольку это единственный оператор сравнения, определенный в нашем классе myArray.

Предположим, что ints — это myArray, а other — объект класса OtherType, который может быть неявно преобразован в myArray. В соответствии с разделом C++ Core Guideline C.86 мы определим operator== как глобальную функцию со следующим прототипом:

```
bool operator==(const MyArray& left, const MyArray& right) noexcept;
```

Это позволит использовать следующие выражения смешанного типа:

```
ints == other
```

или:

```
other == ints
```

Не существует определения операторной функции operator==, которое предоставляло бы параметры, точно соответствующие операндам myArray и OtherType или OtherType и myArray. Однако C++ допускает в каждом выражении только одно преобразование типов, определенное пользователем. Итак, если объекты OtherType могут быть неявно преобразованы в объекты myArray, компилятор преобразует объект other в myArray, а затем вызывает функцию operator==, получающую два объекта myArray.

C++ следует сложному набору правил разрешения перегрузки, чтобы определить, какую функцию надо вызывать в каждом выражении с оператором[1]. Если operator== является функцией класса myArray, то левый операнд должен быть объектом myArray. C++ не будет неявно преобразовывать other в myArray для вызова функции класса, поэтому выражение

```
other == ints
```

с объектом класса OtherType слева вызовет ошибку компиляции. Это может сбить с толку программистов, рассчитывающих на компиляцию такого выражения, поэтому в C++ Core Guideline C.86 рекомендуется использовать функцию operator==, не являющуюся элементом класса myArray.

11.6.11. Перегрузка оператора индекса

В строках 36 и 39 на илл. 11.4:

```
int& operator[](size_t index);
const int& operator[](size_t index) const;
```

[1] «Overload Resolution». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/language/overload_resolution.

объявляются перегруженные операторы индекса (определены в строках 105–112 и 116–123). Когда компилятор видит, например, выражение `ints1[5]`, он вызывает соответствующую перегруженную функцию `operator[]`, генерируя такой вызов:

```
ints1.operator[](5)
```

Компилятор вызывает константную версию `operator[]` (строки 116–123), когда оператор индекса используется для объекта `const myArray`. Например, если вы передаете объект `myArray` в функцию, получающую его как `const myArray&` с именем `z`, то для выполнения такой инструкции требуется константная версия `operator[]`:

```
std::cout << z[3];
```

Запомните: когда объект является константным, программа может вызывать только константные функции этого объекта.

```
103 // Перегруженный оператор индекса для неконстантных объектов MyArray;
104 // возврат ссылки создает изменяемое значение lvalue
105 int& MyArray::operator[](size_t index) {
106    // Проверяем корректность индекса
107    if (index >= m_size) {
108       throw std::out_of_range{"Index out of range"};
109    }
110
111    return m_ptr[index]; // Возвращаем ссылку
112 }
113
114 // Перегруженный оператор индекса для неконстантных объектов MyArray;
115 // возврат константной ссылки создает неизменяемое значение lvalue
116 const int& MyArray::operator[](size_t index) const {
117    // Проверяем корректность индекса
118    if (index >= m_size) {
119       throw std::out_of_range{"Index out of range"};
120    }
121
122    return m_ptr[index]; // Возвращаем копию элемента
123 }
124
```

В каждом определении `operator[]` проверяется, что индекс находится в допустимом диапазоне. Если это условие не выполнено, выдается исключение `out_of_range` (заголовок `<stdexcept>`). Если индекс находится в допустимом диапазоне, неконстантная версия `operator[]` возвращает соответствующий элемент объекта `myArray` в виде ссылки, которая может быть использована в качестве изменяемого значения *lvalue* в левой части выражения присваивания для изменения элемента массива. Константная версия `operator[]` возвращает константную ссылку на соответствующий элемент массива.

Операторы индекса, используемые в строках 111 и 122, принадлежат к классу `unique_ptr`. Когда умный указатель `unique_ptr` управляет массивом в динамически

выделенной памяти, перегруженный для класса `unique_ptr` оператор `[]` позволяет получить доступ к элементам массива.

11.6.12. Перегрузка унарного конвертирующего оператора bool

Для преобразования одних типов данных в другие вы можете определять собственные конвертирующие операторы, которые также называются **перегруженными операторами приведения типов (overloaded cast operators)**. В строке 42 на илл. 11.4

```
explicit operator bool() const noexcept {return size() != 0;}
```

определяется встроенный (inline) перегруженный оператор `bool`, который преобразует объект `myArray` в значение `true`, если объект `myArray` не пустой (то есть содержит хотя бы один элемент), или `false`, если объект пустой. Как и в конструкторах с одним аргументом, мы объявили этот оператор со спецификатором `explicit`, чтобы компилятор не мог использовать его для неявных преобразований типов (подробнее в разделе 11.9). Перегруженные конвертирующие операторы не содержат тип возвращаемого значения слева от ключевого слова `operator`. Возвращаемый тип — это тип конвертирующего оператора, в данном случае `bool`.

На илл. 11.3 в строке 103 мы использовали объект `ints5` в качестве условия инструкции `if`, чтобы определить, содержит ли этот объект элементы. В этом случае C++ вызывал функцию `operator bool` для контекстного преобразования объекта `ints5` в значение `bool`, используемое в качестве условия. Вы также можете вызвать эту функцию явно:

```
static_cast<bool>(ints5)
```

Мы подробнее рассмотрим преобразования между типами в разделе 11.8.

11.6.13. Перегрузка оператора префиксного инкремента

Вы можете перегружать операторы префиксного и постфиксного инкремента и декремента. Концепции, которые мы демонстрируем здесь и в разделе 11.6.14 для операторов `++`, применимы и к операторам `--`. В разделе 11.6.14 объясняется, как компилятор отличает префиксную версию от постфиксной.

В строке 45 на илл. 11.4:

```
MyArray& operator++();
```

объявляется унарный перегруженный оператор префиксного инкремента (`++`). Когда компилятор, например, видит выражение `++ints4`, он вызывает перегруженную функцию префиксного инкремента класса `myArray` (`++`), генерируя такой вызов:

```
ints4.operator++()
```

Это вызывает функцию, определенную в строках 126–132, которая прибавляет 1 к каждому элементу массива таким образом:

- создается объект `span` с именем `items` для представления целочисленного массива в динамической памяти;
- используется алгоритм `for_each` стандартной библиотеки для вызова функции, однократно выполняющей задачу для каждого элемента `span`.

```
125 // Выполняем преинкремент каждого элемента и возвращаем обновленный MyArray
126 MyArray& MyArray::operator++() {
127    // Используем span и for_each для инкремента каждого элемента
128    const std::span<int> items{m_ptr.get(), m_size};
129    std::for_each(std::begin(items), std::end(items),
130       [](auto& item){++item;});
131    return *this;
132 }
133
```

Как и в алгоритме `copy`, первые два аргумента алгоритма `for_each` представляют диапазон обрабатываемых элементов. Третий аргумент — это функция, получающая один аргумент и выполняющая с ним указанную задачу. В нашем случае это лямбда-выражение, которое вызывается один раз для каждого элемента в заданном диапазоне. Поскольку алгоритм `for_each` выполняет внутреннюю итерацию по элементам, он передает текущий элемент в качестве аргумента лямбды (`item`). Затем лямбда выполняет задачу, используя переданное значение. Аргумент лямбды является неконстантной ссылкой (`auto&`), поэтому выражение `++item` в теле лямбды изменяет исходный элемент.

Оператор возвращает ссылку на объект `myArray`, который он только что обработал. Это позволяет использовать обработанный объект в качестве *lvalue* — именно так работает встроенный оператор префиксного инкремента для основных типов.

11.6.14. Перегрузка оператора постфиксного инкремента

Перегрузка оператора постфиксного инкремента представляет проблему. Компилятор должен различать сигнатуры перегруженных функций операторов префиксного и постфиксного инкремента. По соглашению, когда компилятор видит выражение постфиксного инкремента, например `ints4++`, он генерирует такой вызов:

```
ints4.operator++(0)
```

Аргумент `0` является строго **фиктивным значением (dummy value)**, которое компилятор использует, чтобы различать операторные функции префиксного и постфиксного инкремента. Аналогичный синтаксис позволяет различать операторные функции префиксного и постфиксного декремента.

В строке 48 на илл. 11.4

```
MyArray operator++(int);
```

объявляется унарный перегруженный оператор постфиксного инкремента класса `myArray` (`++`) с параметром `int`, который принимает фиктивное значение `0`. Этот параметр не используется, поэтому он объявляется без имени параметра. Для эмуляции эффекта постфиксного инкремента мы должны вернуть неизмененную копию объекта `myArray`. Итак, в определении функции (строки 135–139):

- используется копирующий конструктор класса `myArray` для создания локальной копии исходного объекта;
- вызывается оператор префиксного инкремента для прибавления `1` к каждому элементу объекта `myArray`[1];
- возвращается локальная копия объекта по значению — это еще один случай, когда компиляторы могут использовать оптимизацию именованного возвращаемого значения (NRVO).

Дополнительный локальный объект, созданный оператором постфиксного инкремента (или декремента), может снизить быстродействие, особенно если оператор используется в цикле. По этой причине старайтесь отдавать предпочтение операторам префиксного инкремента и декремента.

```
134 // Выполняем постинкремент каждого элемента и возвращаем исходный MyArray
135 MyArray MyArray::operator++(int) {
136    MyArray temp(*this);
137    ++(*this); // Вызываем префиксный оператор инкремента
138    return temp; // Возвращаем временную копию, сохраненную до инкремента
139 }
140
```

11.6.15. Перегрузка оператора сложения с присваиванием

В строке 51 на илл. 11.4:

```
MyArray& operator+=(int value);
```

для класса `myArray` объявляется перегруженный оператор сложения с присваиванием (`+=`), который прибавляет значение к каждому элементу объекта и возвращает ссылку на измененный объект для возможности реализации каскадных вызовов. Как и в случае с оператором префиксного инкремента, мы используем `span` и алгоритм `for_each` стандартной библиотеки для обработки каждого элемента. В этом случае лямбда-выражение, которое мы передаем в качестве последнего аргумента `for_each` (строка 146), использует параметр `value` из функции `operator+=`. Список захвата лямбда-выражения `[value]` указывает, что компилятор должен разрешить использовать `value` в теле лямбды — это называется **захватом (capturing)** переменной. Мы подробнее поговорим о захватах в главе 14.

[1] Херб Саттер (Herb Sutter), «GotW #2 Solution: Temporary Objects». 13 мая 2013 г. Был доступен: 15 сентября 2023 г. https://herbsutter.com/2013/05/13/gotw-2-solution-temporary-objects/.

```
141 // Прибавляем число к каждому элементу и возвращаем обновленный объект
142 MyArray& MyArray::operator+=(int value) {
143    // Используем span и for_each для инкремента каждого элемента
144    const std::span<int> items{m_ptr.get(), m_size};
145    std::for_each(std::begin(items), std::end(items),
146       [value](auto& item) {item += value;});
147    return *this;
148 }
149
```

11.6.16. Перегрузка бинарных операторов извлечения из потока и вставки в поток

Вы можете вводить и выводить данные основных типов, используя оператор извлечения из потока (>>) и оператор вставки в поток (<<). В стандартной библиотеке C++ эти операторы перегружены для каждого из основных типов, включая указатели и строки. Вы можете перегрузить эти операторы для ввода и вывода пользовательских типов.

В строке 10 на илл. 11.4

```
friend std::istream& operator>>(std::istream& in, MyArray& a);
```

и в строке 58 на илл. 11.4

```
std::ostream& operator<<(std::ostream& out, const MyArray& a);
```

объявляются перегруженный оператор извлечения из потока (>>) и перегруженный оператор вставки в поток (<<), не являющиеся элементами класса. Мы объявили функцию `operator>>` как дружественную, потому что она будет напрямую обращаться к скрытым данным класса `MyArray`, что повысит быстродействие. Функция `operator<<` не объявлена как дружественная. Как вы увидите, она вызывает функцию `toString` класса `MyArray` для получения строкового представления объекта и последующего вывода.

Реализация оператора извлечения из потока

Перегруженный оператор извлечения из потока (>>) (строки 152–160) принимает в качестве аргументов две ссылки: на объект `istream` и на объект `myArray`. Он возвращает ссылку на объект `istream`, чтобы обеспечить каскадирование ввода, например:

```
std::cin >> ints1 >> ints2;
```

Когда компилятор видит, например, выражение `cin >> ints1`, он вызывает функцию `operator>>`, не являющуюся элементом класса:

```
operator>>(std::cin, ints1)
```

Мы сразу поясним, почему эта функция не должна находиться внутри класса. Когда вызов функции завершается, он возвращает ссылку на объект `cin`, кото-

рая будет использоваться в инструкции `cin` для ввода значений в объект `ints2`. Функция создает `span` (строка 153), представляющий целочисленный массив в динамически выделяемой памяти, и в строках 155–157 перебирает элементы, считывая по одному значению за раз из входного потока и помещая это значение в соответствующий элемент массива `int` в динамически выделяемой памяти.

```
150 // Перегруженный оператор извлечения из потока для класса MyArray;
151 // вводит все значения в MyArray
152 std::istream& operator>>(std::istream& in, MyArray& a) {
153    std::span<int> items{a.m_ptr.get(), a.m_size};
154
155    for (auto& item : items) {
156       in >> item;
157    }
158
159    return in; // Позволяет осуществить cin >> x >> y;
160 }
161
```

Реализация оператора вставки в поток

Перегруженная функция вставки в поток (`<<`) (строки 163–166) принимает ссылку на `ostream` и константную ссылку на объект `myArray` в качестве аргументов и возвращает ссылку на `ostream`. Перегруженная функция вызывает функцию `toString` класса `myArray` и выводит результат вызова.

Функция возвращает ссылку на `ostream`, чтобы обеспечить инструкции каскадного вывода, например:

```
std::cout << ints1 << ints2;
```

Когда компилятор видит, например, выражение `std::cout << ints1`, он вызывает функцию `operator<<`, не являющуюся элементом класса:

```
operator<<(std::cout, ints1)
```

```
162 // Перегруженный оператор вставки в поток для класса MyArray;
163 std::ostream& operator<<(std::ostream& out, const MyArray& a) {
164    out << a.toString();
165    return out; // Позволяет осуществить std::cout << x << y;
166 }
167
```

Почему operator>> и operator<< не должны быть функциями класса

Функции `operator>>` и `operator<<` определены как функции, не являющиеся элементами класса, поэтому мы можем указать порядок их операндов в списке параметров каждой функции. В бинарном перегруженном операторе, реализованном как сторонняя функция, первым параметром является левый операнд, а вторым — правый операнд.

Для операторов >> и << объект класса myArray должен быть правым операндом каждого оператора, чтобы сохранить привычную для программистов C++ форму записи, например:

```
std::cin >> ints4;
std::cout << ints4;
```

Если бы мы определили эти функции внутри класса myArray, программистам пришлось бы писать странно выглядящие инструкции для ввода и вывода пользовательских объектов:

```
ints4 >> std::cin;
ints4 << std::cout;
```

Такие инструкции вводили бы в заблуждение, иногда приводя к ошибкам компиляции. Программисты, работающие с объектами cin и cout, знают, что их имена всегда ставятся слева от этих операторов.

Перегруженные бинарные операторы могут быть функциями только класса левого операнда. Чтобы operator>> и operator<< могли быть функциями нашего пользовательского класса, нам пришлось бы модифицировать классы istream и ostream стандартной библиотеки, что недопустимо.

Выбор между функциями классов и свободными функциями

Перегруженные операторные функции, как и функции вообще, могут быть:

- элементами классов с прямым доступом к деталям внутренней реализации классов;
- дружественными функциями с прямым доступом к деталям внутренней реализации классов;
- **свободными функциями (free functions)**, то есть функциями, которые не являются ни элементами какого-либо класса, ни дружественными функциями; они взаимодействуют с объектами класса через его открытые интерфейсы.

В C++ Core Guidelines отмечается, что функция должна быть элементом класса только при необходимости прямого доступа к его внутренним деталям реализации, таким как скрытые данные[1].

Другой причиной использования свободных функций является определение **коммутативных операторов (commutative operators)**. Рассмотрим класс HugeInt для целых чисел произвольного размера. С объектом bigInt класса HugeInt мы можем выполнить такие выражения:

```
bigInt + 7
7 + bigInt
```

[1] C++ Core Guidelines, «C.4: Make a Function a Member Only If It Needs Direct Access to the Representation of a Class». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-member.

Каждое из них суммирует целое число и объект класса `HugeInt`. Подобно встроенному оператору + для основных типов, каждое выражение будет создавать временный объект `HugeInt`, содержащий сумму. Для поддержки таких выражений следует определить две версии `operator+`, обычно в виде свободных функций:

```
friend HugeInt operator+(const HugeInt& left, int right);
friend HugeInt operator+(int left, const HugeInt& right);
```

Чтобы избежать дублирования кода, вторая функция, как правило, вызывает первую.

11.6.17. Дружественная функция swap

В строке 13 на илл. 11.4:

```
friend void swap(MyArray& a, MyArray& b) noexcept;
```

объявляется функция `swap`, используемая оператором копирующего присваивания для реализации идиомы копирования и обмена. Функция объявлена как `noexcept`: она не должна завершаться сбоем, так как обмен содержимым двух существующих объектов не создает новых ресурсов[1]. Эта функция (строки 169–172) получает два объекта `MyArray` и вызывает функцию `swap` стандартной библиотеки для обмена содержимым элементов данных `m_size` каждого из объектов. Она также использует функцию `swap` объекта `unique_ptr` для обмена содержимым элементов данных `m_ptr` каждого из объектов.

```
168 // Используем функцию swap для реализации идиомы копирования и обмена
169 void swap(MyArray& a, MyArray& b) noexcept {
170    std::swap(a.m_size, b.m_size); // Вызываем функцию std::swap
171    a.m_ptr.swap(b.m_ptr); // Вызываем функцию swap класса unique_ptr
172 }
```

11.7. Оператор трехстороннего сравнения (C++20)

Вы часто будете сравнивать объекты пользовательских типов. Например, для сортировки объектов в порядке возрастания или убывания с помощью функции `sort` стандартной библиотеки (раздел 6.12; подробнее в главе 14) объекты должны быть сравнимыми. Чтобы поддерживать операции сравнения, вы можете перегрузить операторы равенства (== и !=) и сравнения (<, <=, > и >=) для ваших классов. Обычной практикой является определение операторных функций < и == и последующая перегрузка операторов !=, <=, > и >= на основе операторных функций < и ==. Например, для класса `Time`, представляющего

[1] C++ Core Guidelines, «C.84: A swap Function May Not Fail». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-swap-fail.

время суток, оператор <= может быть реализован как встроенная функция с использованием оператора <:

```
bool operator<=(const Time& right) const {
   return !(right < *this);
}
```

Это приводит к большому количеству «шаблонного» кода, в котором единственное различие в определениях перегруженных операторов !=, <=, > и >= между классами — это типы аргументов.

Для большинства ваших типов компилятор может обрабатывать операторы сравнения с помощью сгенерированной компилятором по умолчанию реализации **трехстороннего оператора сравнения (three-way comparison operator)** <=>[1, 2, 3] (заголовок <compare>, C++20). Этот оператор выглядит так, что программисты иногда называют его «звездолетом» («spaceship»)[4]. На илл. 11.6 показано применение оператора <=> к классу Time (строки 8–23), содержащему:

- конструктор (строки 10–11) для инициализации скрытых элементов данных (строки 20–22);
- функцию toString (строки 13–15) для создания строкового представления объекта;
- определение перегруженного оператора <=> со спецификатором default (строка 18).

Сгенерированный компилятором по умолчанию оператор <=> работает с любым классом, все элементы данных которого поддерживают операторы равенства и сравнения. Кроме того, для классов, содержащих традиционные массивы в качестве элементов данных, компилятор применяет перегруженный оператор <=> поэлементно, сравнивая пары объектов, имеющих тип элементов этих массивов.

```
1 // fig11_06.cpp
2 // Оператор трехстороннего сравнения (C++20)
3 #include <compare>
4 #include <fmt/format.h>
5 #include <iostream>
6 #include <string>
```

[1] «C++ Russia 2018: Herb Sutter, New in C++20: The Spaceship Operator». 25 июня 2018 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=ULkwKsag0Yk.

[2] Сай Бренд (Sy Brand), «Spaceship Operator». 23 августа 2018 г. Был доступен: 15 сентября 2023 г. https://blog.tartanllama.xyz/spaceship-operator/.

[3] Кэмерон Да Камара (Cameron DaCamara), «Simplify Your Code with Rocket Science: C++20's Spaceship Operator». 27 июня 2019 г. Был доступен: 15 сентября 2023 г. https://devblogs.microsoft.com/cppblog/simplifyyour-code-with-rocket-science-c20s-spaceship-operator/.

[4] Название «звездолет» придумал Рэндал Л. Шварц, когда рассказывал студентам об аналогичном операторе в курсе программирования на Perl. Оператор напомнил ему космический корабль в ранней видеоигре. https://groups.google.com/a/dartlang.org/g/misc/c/WS5xftItpl4/m/jcIttrMq8agJ?pli=1.

```
 7
 8 class Time {
 9 public:
10    Time(int hr, int min, int sec) noexcept
11       : m_hr{hr}, m_min{min}, m_sec{sec} {}
12
13    std::string toString() const {
14       return fmt::format("hr={}, min={}, sec={}", m_hr, m_min, m_sec);
15    }
16
17    // Оператор <=> автоматически поддерживает операторы равенства и сравнения
18    auto operator<=>(const Time& t) const noexcept = default;
19 private:
20    int m_hr{0};
21    int m_min{0};
22    int m_sec{0};
23 };
24
25 int main() {
26    const Time t1(12, 15, 30);
27    const Time t2(12, 15, 30);
28    const Time t3(6, 30, 0);
29
30    std::cout << fmt::format("t1: {}\nt2: {}\nt3: {}\n\n",
31                t1.toString(), t2.toString(), t3.toString());
32
```

```
t1: hr=12, min=15, sec=30
t2: hr=12, min=15, sec=30
t3: hr=6, min=30, sec=0
```

Илл. 11.6 | Оператор трехстороннего сравнения (C++20)

В функции `main` в строках 26–28 создаются три объекта класса `Time` (мы будем их использовать в различных операциях сравнения) и выводятся их строковые представления. Объекты `t1` и `t2` представляют время 12:15:30 PM, а объект `t3` — 6:30:00 AM. Остальная часть этой программы разделена на фрагменты для удобства обсуждения.

С помощью оператора <=> компилятор поддерживает все операторы сравнения

Когда вы позволяете компилятору генерировать перегруженный оператор трехстороннего сравнения (`<=>`), ваш класс поддерживает все операторы равенства и сравнения, которые мы демонстрируем в строках 34–46. В каждом случае компилятор переписывает выражение, подобное

```
t1 == t2
```

в выражение с оператором трехстороннего сравнения (`<=>`):

```
(t1 <=> t2) == 0
```

Результатом выражения `t1 <=> t2` является `0`, если объекты равны; или отрицательное значение, если `t1` меньше `t2`; или положительное значение, если `t1` больше `t2`. Как вы можете видеть, код в строках 34–46 скомпилировался и выдал правильные значения в выходных данных, хотя в классе `Time` не определены никакие операторы равенства или сравнения.

```
33    // Операторы равенства и сравнения
34    std::cout << fmt::format("t1 == t2: {}\n", t1 == t2);
35    std::cout << fmt::format("t1 != t2: {}\n", t1 != t2);
36    std::cout << fmt::format("t1 < t2: {}\n", t1 < t2);
37    std::cout << fmt::format("t1 <= t2: {}\n", t1 <= t2);
38    std::cout << fmt::format("t1 > t2: {}\n", t1 > t2);
39    std::cout << fmt::format("t1 >= t2: {}\n\n", t1 >= t2);
40
41    std::cout << fmt::format("t1 == t3: {}\n", t1 == t3);
42    std::cout << fmt::format("t1 != t3: {}\n", t1 != t3);
43    std::cout << fmt::format("t1 < t3: {}\n", t1 < t3);
44    std::cout << fmt::format("t1 <= t3: {}\n", t1 <= t3);
45    std::cout << fmt::format("t1 > t3: {}\n", t1 > t3);
46    std::cout << fmt::format("t1 >= t3: {}\n\n", t1 >= t3);
47
```

```
t1 == t2: true
t1 != t2: false
t1 < t2: false
t1 <= t2: true
t1 > t2: false
t1 >= t2: true

t1 == t3: false
t1 != t3: true
t1 < t3: false
t1 <= t3: false
t1 > t3: true
t1 >= t3: true
```

Явное использование оператора <=>

Вы можете явно использовать оператор `<=>` в выражениях. Результат выражения с оператором `<=>` не преобразуется в значение `bool`, поэтому вы должны сравнить его с `0`, чтобы использовать оператор трехстороннего сравнения в условии, как показано в строках 49, 53 и 57.

```
48    // Используем <=> в операциях сравнения
49    if ((t1 <=> t2) == 0) {
50       std::cout << "t1 is equal to t2\n";
51    }
52
53    if ((t1 <=> t3) > 0) {
54       std::cout << "t1 is greater than t3\n";
```

```
55    }
56
57    if ((t3 <=> t1) < 0) {
58       std::cout << "t3 is less than t1\n";
59    }
60 }
```

```
t1 is equal to t2
t1 is greater than t3
t3 is less than t1
```

11.8. Преобразования между типами

Большинство программ работает со многими типами данных. Иногда все операции реализуются в пределах одного и того же типа. Например, сложение числа `int` с числом `int` дает в результате число `int`. Однако часто требуется преобразовать (конвертировать) данные одного типа в данные другого типа. Это может происходить при присваиваниях и вычислениях, при передаче значений в функции и при возврате значений из функций. Компилятор знает, как выполнять преобразования между основными типами данных. И вы тоже можете использовать операторы приведения типов для явных преобразований между основными типами.

А если речь идет о пользовательских типах? Компилятор не знает, как конвертировать одни пользовательские типы в другие или пользовательские типы в основные. Вы должны определить реализации таких преобразований. Их можно выполнять с помощью **конвертирующих конструкторов (conversion constructors)**, вызываемых с одним аргументом (мы называем их одноаргументными конструкторами). Они могут преобразовывать объекты других типов (включая основные типы) в объекты нужного вам класса.

Конвертирующие операторы

Конвертирующий оператор (conversion operator), также называемый **оператором приведения типов (cast operator)**, может конвертировать тип объекта класса, то есть преобразовать его в другой тип. Конвертирующий оператор должен быть нестатической функцией класса. В примере с классом `MyArray` мы реализовали перегруженный конвертирующий оператор, который приводил тип `MyArray` к значению `bool`, показывающему, содержит ли объект элементы.

Неявные вызовы конвертирующих операторов и конвертирующих конструкторов

Особенностью конвертирующих операторов и конвертирующих конструкторов является возможность их неявного вызова компилятором для создания объектов. Предположим, что объект класса `myArray` появляется в инструкции, где ожидается значение `bool`:

```
if (ints1) { // Инструкция if ожидает условие
   ...
}
```

Здесь компилятор может вызвать перегруженную операторную функцию `operator bool`, чтобы преобразовать объект в значение `bool`, которое можно использовать в качестве условия.

Когда для неявного преобразования используется конвертирующий конструктор или конвертирующий оператор, C++ может применить в одном выражении только один неявный вызов функции такого конструктора или оператора (то есть не более одного пользовательского преобразования), стараясь привести объект в соответствие с выражением. Компилятор не будет выполнять для этой цели целый ряд неявных пользовательских преобразований.

11.9. Конструкторы и конвертирующие операторы со спецификатором explicit

Напомним, что мы объявляли со спецификатором `explicit` каждый конструктор, который может быть вызван с одним аргументом, в том числе конструкторы с большим количеством параметров, но с аргументами по умолчанию. За исключением копирующих и перемещающих конструкторов, все конструкторы без спецификатора `explicit`, которые можно вызвать с одним аргументом, могут быть использованы компилятором для неявных преобразований типов. Аргумент конструктора преобразуется в объект класса, в котором определен конструктор. Типы конвертируются автоматически, вам не надо выполнять явное приведение типов.

В некоторых ситуациях неявные преобразования типов нежелательны или приводят к ошибкам. Например, в классе `myArray` определен конструктор, принимающий один аргумент типа `size_t`. Целью этого конструктора является создание объекта `myArray`, содержащего указанное количество элементов. Но если бы этот конструктор не был объявлен как `explicit`, компилятор мог бы использовать его для неявных преобразований.

К сожалению, компилятор иногда может выполнять неявные преобразования в таких местах кода, где вы не ожидаете. Это приводит к логическим ошибкам во время выполнения программы или к двусмысленным выражениям, порождающим ошибки компиляции.

Случайное использование конструктора с одним аргументом в качестве конвертирующего конструктора

На илл. 11.7 показана программа, использующая класс `myArray` из раздела 11.6 для демонстрации ошибочного неявного преобразования. Чтобы сделать возможным неявное преобразование, мы удалили ключевое слово `explicit` из строки 16 в файле `myArray.h` (илл. 11.4).

```
1 // fig11_07.cpp
2 // Одноаргументные конструкторы и неявные преобразования
3 #include <iostream>
4 #include "MyArray.h"
5 using namespace std;
6
7 void outputArray(const MyArray&); // Прототип
8
9 int main() {
10    MyArray ints1(7); // 7-элементный MyArray
11    outputArray(ints1); // Выводим объект ints1 типа MyArray
12    outputArray(3); // Конвертируем 3 в тип MyArray и выводим на экран
13 }
14
15 // Выводим содержимое MyArray
16 void outputArray(const MyArray& arrayToOutput) {
17    std::cout << "The MyArray received has " << arrayToOutput.size()
18       << " elements. The contents are: " << arrayToOutput << "\n";
19 }
```

```
MyArray(size_t) constructor
The MyArray received has 7 elements. The contents are: {0, 0, 0, 0, 0, 0, 0}
MyArray(size_t) constructor
The MyArray received has 3 elements. The contents are: {0, 0, 0}
MyArray destructor
MyArray destructor
```

Илл. 11.7 | Одноаргументные конструкторы и неявные преобразования

В строке 10 (илл. 11.7) создается объект `ints1` и вызывается конструктор с одним аргументом `7`, указывающим количество элементов в объекте `ints1`. Напомним, что конструктор класса `myArray`, получающий аргумент `size_t`, инициализирует все элементы объекта значением `0`. В строке 11 вызывается функция `outputArray` (определенная в строках 16–19), получающая в качестве аргумента константную ссылку (`const MyArray&`). Эта функция выводит количество элементов и содержимое своего аргумента. В этом случае размер объекта `MyArray` равен `7`, поэтому функция `outputArray` выводит семь нулей.

В строке 12 вызывается функция `outputArray` со значением `3` в качестве аргумента. В этой программе нет функции `outputArray`, принимающей аргумент `int`. Поэтому компилятор выясняет, есть ли возможность преобразования аргумента в объект `myArray`. Поскольку класс `myArray` предоставляет конструктор с одним аргументом `size_t` (который может принимать значение `int`) и этот конструктор не объявлен как `explicit`, компилятор предполагает, что конструктор является конвертирующим, и использует его для преобразования аргумента во временный объект `myArray`, содержащий три элемента. Затем компилятор передает этот временный объект `myArray` в функцию `outputArray`, которая выводит размер и содержимое временного объекта. Таким образом, даже если мы явно не предоставляем функцию `outputArray`, принимающую аргумент типа `int`, компилятор может

скомпилировать строку 12. Выходные данные показывают, что `myArray` содержит три элемента со значением `0`.

Предотвращение неявных преобразований в конструкторах с одним аргументом

Мы объявили каждый одноаргументный конструктор со спецификатором `explicit`, чтобы запретить неявные преобразования типов, которые в нашем случае недопустимы. Конструктор, объявленный как `explicit`, не может быть использован для неявного преобразования типов.

В нашей следующей программе используется класс `myArray` из раздела 11.6, содержащий спецификатор `explicit` в объявлении одноаргументного конструктора, получающего `size_t`:

```
explicit MyArray(size_t size);
```

На илл. 11.8 показана измененная версия программы с илл. 11.7. Когда она компилируется, компилятор g++ выдает сообщение об ошибке:

```
error: invalid initialization of reference of type 'const MyArray&'
from expression of type 'int'
```

Оно означает, что целое число, переданное в качестве аргумента в функцию `outputArray` в строке 12, не может быть преобразовано в `const myArray&`. В строке 13 демонстрируется, как конструктор со спецификатором `explicit` можно использовать для создания временного объекта с тремя элементами и для передачи его в функцию `outputArray`.

```
 1 // fig11_08.cpp
 2 // Демонстрация конструктора со спецификатором explicit
 3 #include <iostream>
 4 #include "MyArray.h"
 5 using namespace std;
 6
 7 void outputArray(const MyArray&); // Прототип
 8
 9 int main() {
10    MyArray ints1{7}; // 7-элементный MyArray
11    outputArray(ints1); // Выводим объект ints1 типа MyArray
12    outputArray(3); // Конвертируем 3 в тип MyArray и выводим на экран
13    outputArray(MyArray(3)); // Вызов одноаргументного explicit-конструктора
14 }
15
16 // Выводим содержимое MyArray
17 void outputArray(const MyArray& arrayToOutput) {
18    std::cout << "The MyArray received has " << arrayToOutput.size()
19       << " elements. The contents are: " << arrayToOutput << "\n";
20 }
```

Илл. 11.8 | Демонстрация конструктора со спецификатором explicit

Вы должны всегда использовать ключевое слово `explicit` в одноаргументных конструкторах, которые не предназначены для неявных преобразований в качестве конвертирующих конструкторов. Если же вы хотите разрешить неявные преобразования с помощью вашего одноаргументного конструктора, покажите это пользователям вашего класса, объявив конструктор со спецификатором `explicit(false)`. Это нововведение появилось в C++20.

Конвертирующие операторы со спецификатором explicit (C++11)

Вы можете объявлять со спецификатором `explicit` (или `explicit(true)` в C++20) не только одноаргументные конструкторы, но и конвертирующие операторы, чтобы компилятор не мог использовать их для неявных преобразований. Например, в классе `myArray` прототип:

```
explicit operator bool() const noexcept;
```

объявляет конвертирующий оператор как `explicit`, поэтому вам надо вызывать его явно с помощью `static_cast`:

```
static_cast<bool>(myArrayObject)
```

Как мы показывали в примере с классом `myArray`, C++ по-прежнему может выполнять контекстные преобразования, используя конвертирующий оператор `explicit bool`, чтобы преобразовать объект в значение `bool` в условии.

11.10. Перегрузка оператора вызова функции

Перегрузка оператора вызова функции `()` — это очень мощное средство, позволяющее функциям принимать произвольное количество параметров, разделенных запятыми. Мы продемонстрируем перегруженный оператор вызова функции в разделе 14.5.

11.11. Итоги

В этой главе мы рассмотрели создание уникальных пользовательских классов и перегрузку операторов, позволяющую обрабатывать объекты этих классов стандартными операторами C++. Сначала мы использовали несколько готовых перегруженных операторов класса `string`. Далее мы показали, как перегружать операторы, какие операторы могут быть перегружены и каковы правила и ограничения их перегрузки.

Вы познакомились с управлением динамической памятью с помощью операторов `new` и `delete`, которые во время выполнения программы выделяют и освобождают память для объектов и для традиционных массивов на основе указателей. Мы обсудили проблемы, возникающие при работе с операторами `new` и `delete`, когда программист забывает освободить память, которая ему больше не нужна. Мы представили концепцию RAII («получение ресурсов есть инициализация») и умные указатели. Вы научились динамически выделять память с помощью умных

указателей `unique_ptr`, которые предотвращают утечки памяти, автоматически освобождая память, когда объекты покидают область видимости.

Далее мы создали класс `myArray`, в котором благодаря перегруженным операторам и другим решениям отсутствуют фатальные проблемы массивов в стиле C. Мы реализовали в нем пять специальных функций, обычно определяемых в классах, управляющих собственными ресурсами: копирующий конструктор, оператор копирующего присваивания, перемещающий конструктор, оператор перемещающего присваивания и деструктор. Мы также обсудили автоматическую генерацию специальных функций с помощью обозначения `= default` и их удаление с помощью обозначения `= delete`. Класс `myArray` перегружает много операторов и предоставляет возможность преобразования типа `myArray` в `bool`, чтобы клиентский код мог определить, является ли объект `myArray` пустым или содержит элементы.

Мы представили оператор трехстороннего сравнения `<=>`, введенный в C++20. Вы узнали, что сгенерированный компилятором по умолчанию оператор `<=>` позволяет некоторым классам поддерживать все шесть операторов равенства и сравнения без необходимости их явной перегрузки. Мы подробно обсудили преобразования между типами, проблемы с неявными преобразованиями в одноаргументных конструкторах и предотвращение этих проблем с помощью ключевого слова `explicit`. Затем мы упомянули о перегрузке оператора вызова функции `()`; далее мы вернемся к этой теме.

Вы уже знакомы с исключениями. В следующей главе подробно рассматривается обработка исключений. Она позволит вам создавать более надежные и отказоустойчивые программы, способные справиться с неожиданной проблемой и продолжить работу, а в крайнем случае — корректно завершиться. Мы покажем, как обрабатывать исключения, возникающие, когда оператору `new` не удается выделить память для объекта. Мы представим несколько классов обработки исключений, имеющихся в стандартной библиотеке C++, и продемонстрируем создание пользовательского класса обработки исключений.

12

Исключения и обзор контрактного программирования

В этой главе вы:

- познакомитесь с обработкой исключений и с инструкциями `try`, `catch` и `throw`;
- предоставите гарантии безопасности исключений для своего кода;
- разберетесь в иерархии исключений стандартной библиотеки;
- определите пользовательский класс исключений;
- узнаете, как размотка стека позволяет перехватывать исключения, появившиеся в другой области видимости;
- научитесь обрабатывать исключения, возникшие при динамическом выделении памяти;
- научитесь перехватывать исключения любого типа с помощью инструкции `catch(...)`;
- узнаете, что происходит с исключениями, которые не были перехвачены;
- узнаете, какие исключения не надо или нельзя обрабатывать;
- поймете, почему в некоторых организациях запрещены исключения и как это влияет на сложность разработки программ;
- встретитесь с проблемой снижения быстродействия из-за обработки исключений;
- познакомитесь с контрактами, которые устраняют необходимость во многих исключениях, позволяя вам объявлять больше функций со спецификатором `noexcept`.

12.1. Введение

На C++ пишут программное обеспечение, широко применяемое в реальном мире и критически важное для бизнеса. Создатель C++ Бьерн Страуструп ведет на своем веб-сайте список[1], включающий около 150 приложений и систем, частично или полностью написанных на C++. Вот лишь некоторые из них:

- фрагменты большинства основных операционных систем, таких как Apple macOS и Microsoft Windows;
- все компиляторы, с которыми мы работаем в этой книге: GNU g++, Clang и Visual C++;
- Amazon.com;
- финансовые информационные системы Bloomberg, работающие в режиме реального времени;
- многие графические, мультимедийные и авторинговые приложения Adobe;
- различные системы управления базами данных, например MongoDB и MySQL;
- многие проекты NASA, включая фрагменты программного обеспечения марсоходов;

и многое другое.

Еще один обширный список, содержащий более 100 приложений и систем, вы найдете на сайте «The Programming Languages Beacon»[2].

Многие из этих систем крайне сложны. Вот некоторые статистические данные из «Codebases: Millions of Lines of Code Infographic» (эти материалы относятся не только к C++)[3]:

- Бортовое оборудование самолета Boeing 787 вместе с системами онлайн-поддержки содержит 6.5 миллиона строк кода, а все программное обеспечение, обеспечивающее полеты, — 14 миллионов строк кода.
- Истребитель F-35 содержит 24 миллиона строк кода.
- Большой адронный коллайдер в Швейцарии (крупнейший в мире ускоритель элементарных частиц)[4] содержит 50 миллионов строк кода.

---

[1] Бьерн Страуструп (Bjarne Stroustrup), «C++ Applications», 27 октября 2020 г. Был доступен: 15 сентября 2023 г. https://www.stroustrup.com/applications.html.

[2] Винсент Лекстрейт (Vincent Lextrait), «The Programming Languages Beacon v16», март 2016 г. Был доступен: 15 сентября 2023 г. https://www.mentofacturing.com/vincent/implementations.html.

[3] «Codebases: Millions of Lines of Code Infographic», 24 сентября 2015 г. Был доступен: 15 сентября 2023 г. https://www.informationisbeautiful.net/visualizations/million-lines-of-code/. Таблица с источниками данных инфографики: http://bit.ly/CodeBasesInfographicData.

[4] «The Large Hadron Collider». Был доступен: 15 сентября 2023 г. https://home.cern/science/accelerators/large-hadron-collider.

- Типичный современный автомобиль представительского класса содержит 100 миллионов строк кода, и ожидается, что к 2030 году в нем будет 300 миллионов строк кода[1], выполняемых на сотнях процессоров и контроллеров[2].

Ожидается, что для беспилотного автомобиля потребуется миллиард строк кода[3]. В больших и маленьких проектах важно устранять ошибки во время разработки и решать проблемы, которые могут возникнуть после развертывания программного обеспечения в реальных условиях, что и обсуждается в этой главе.

Исключения и их обработка

Как вы уже знаете, исключение — это индикатор проблемы, возникшей во время выполнения программы. Исключения могут возникать:

- явно в блоке `try`;
- при вызове других функций (в том числе библиотечных);
- при ошибочных операциях, например когда оператор `new` не может выделить дополнительную память во время выполнения программы (раздел 12.8).

Обработка исключений поможет вам писать надежные, отказоустойчивые программы, способные обнаружить проблему и затем, в зависимости от обстоятельств:

- решить ее и продолжить выполнение;
- если данное исключение не может или не должно обрабатываться, освободить ресурсы и корректно завершить работу;
- если возникло непредусмотренное исключение, сразу же завершить работу — эту концепцию, называемую немедленным завершением (fail-fast), мы обсуждаем в разделе 12.6.2.

Обработка исключений, размотка стека и повторная генерация исключений

Вы уже знаете, как создаются исключения (раздел 6.15) и как инструкции `try... catch` используются для их обработки (раздел 9.7.11). В этой главе мы подробнее рассмотрим этот способ обработки исключений на примере, который демонстрирует потоки управления:

- при успешном выполнении программы;
- при возникновении исключения.

---

[1] Энтони Мартин (Anthony Martin), «Vehicle Cybersecurity: Control the Code, Control the Road», 18 марта 2020 г. Был доступен: 15 сентября 2023 г. https://www.vehicledynamicsinternational.com/features/vehicle-cybersecurity-control-the-code-control-the-road.html.

[2] Ответ Ференца Валенты (Ferenc Valenta) на вопрос «How many lines of code are in a car?» 10 января 2019 г. Был доступен: 15 сентября 2023 г. https://www.quora.com/How-many-lines-of-code-are-in-a-car/answer/Ferenc-Valenta.

[3] «Jaguar Land Rover Finds the Teenagers Writing the Code for a Self-Driving Future», 15 апреля 2019 г. Был доступен: 15 сентября 2023 г. https://media.jaguarlandrover.com/news/2019/04/jaguar-landrover-finds-teenagers-writing-code-self-driving-future.

Мы рассмотрим варианты перехвата и повторной выдачи исключения и покажем, как C++ обрабатывает исключение, которое не было перехвачено в определенной области видимости. Мы кратко расскажем, как протоколировать исключения в файле или базе данных, чтобы потом изучать эту информацию для отладки приложения.

Когда нужны исключения и какими бывают гарантии безопасности

Исключения предназначены для обработки лишь некоторых видов ошибок, поэтому мы объясним, когда их надо использовать, а когда не надо. Мы также расскажем о нескольких вариантах гарантий безопасности исключений, которые вы можете предоставить в своем коде, начиная с полного отсутствия исключений, которое обозначается спецификатором `noexcept`.

Исключения в конструкторах и деструкторах

Мы обсудим, почему исключения используются для выявления ошибок во время выполнения конструкторов и почему деструкторы не должны генерировать исключения. Мы покажем, как использовать функциональные блоки `try` для перехвата исключений из списка инициализаторов конструктора.

Обработка ошибок выделения динамической памяти

По умолчанию оператор `new` генерирует исключение `bad_alloc` при возникновении ошибки выделения динамической памяти. Мы рассмотрим перехват этого исключения и покажем, как обрабатывались ошибки выделения памяти в устаревшем коде до появления исключения `bad_alloc`.

Иерархия исключений стандартной библиотеки и пользовательские классы исключений

Мы рассмотрим иерархию классов исключений стандартной библиотеки C++ и создадим пользовательский класс, производный от одного из этих классов. Вы поймете, почему важно перехватывать исключения по ссылке, давая возможность обработчикам исключений перехватывать исключения с типами производных классов.

Исключения используются не повсеместно

Большинство функций C++ написано по принципу нулевого влияния на быстродействие. Это означает, что функция не расходует ресурсы, когда вы ее не используете[1]. Исключения нарушают этот принцип: программы с обработкой исключений расходуют больше памяти. Некоторые организации запрещают обработку исключений по этой и другим причинам. Мы обсудим, почему некоторые

[1] Херб Саттер (Herb Sutter), «De-fragmenting C++: Making Exceptions and RTTI More Affordable and Usable», 23 сентября 2021 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=ARYP83yNAWk.

библиотеки предоставляют двойные интерфейсы, позволяя разработчикам выбирать между двумя версиями каждой функции: с исключениями и с кодами ошибок.

Основы контрактного программирования

Эта глава завершается рассмотрением основ программирования с контрактами. Первоначально планировалось добавить контракты в C++20, но потом их перенесли в следующую версию. Мы расскажем о предусловиях, постусловиях и утверждениях, которые реализованы в виде контрактов, проверяемых во время выполнения программы. Если какое-то из условий не выполняется, происходит нарушение контракта. В этом случае по умолчанию программа немедленно завершается, что позволяет быстрее находить ошибки, устранять их во время разработки и, как мы надеемся, создавать более надежный код. Вы сможете протестировать пример кода с контрактами, используя экспериментальную реализацию контрактов GCC на сайте https://godbolt.org.

Исключения отделяют обработку ошибок от логики программы

Обработка исключений — это стандартный механизм обработки ошибок, особенно важный при работе над большими проектами. Как вы увидите, инструкции `try...catch` позволяют разделять успешный путь выполнения и путь выполнения с ошибкой[1, 2], что упрощает чтение и обслуживание вашего кода[3]. Кроме того, если возникает исключение, его нельзя игнорировать. В разделе 12.5 вы увидите, что игнорирование исключения может привести к завершению программы[4, 5].

Индикация ошибок с помощью возвращаемых значений

Без обработки исключений функции, как правило, вычисляют и возвращают значение при успешном выполнении или возвращают индикатор ошибки при сбое:

- Проблема этой архитектуры заключается в том, что программа использует возвращенное значение в последующих вычислениях, предварительно не проверив, является ли оно индикатором ошибки.

---

[1] «Technical Report on C++ Performance», 15 февраля 2006 г. Был доступен: 15 сентября 2023 г. http://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf (страница 34).

[2] «What Does It Mean That Exceptions Separate the Good Path (or Happy Path) from the Bad Path?». Был доступен: 15 сентября 2023 г. https://isocpp.org/wiki/faq/exceptions#exceptions-separate-good-and-bad-path.

[3] Барбара Томпсон (Barbara Thompson), «C++ Exception Handling: Try, Catch, Throw Example», обновлено 1 января 2022 г. Был доступен: 15 сентября 2023 г. https://www.guru99.com/cpp-exceptions-handling.html.

[4] «Technical Report on C++ Performance», 15 февраля 2006 г. Был доступен: 15 сентября 2023 г. http://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf (страница 34).

[5] Манодж Пиюмал (Manoj Piyumal), «Some Useful Facts to Know Before Using C++ Exceptions», 5 декабря 2017 г. Был доступен: 15 сентября 2023 г. https://dzone.com/articles/some-useful-facts-to-know-when-using-c-exceptions.

- Есть случаи, когда возврат кодов ошибок невозможен, например в конструкторах и перегруженных операторах.

Обработка исключений решает эти проблемы.

12.2. Поток управления при обработке исключения

В этом разделе мы продемонстрируем поток управления:

- в случае, если программа выполняется успешно;
- в случае, если возникает исключение.

В целях демонстрации на илл. 12.1–12.2 показана обычная арифметическая проблема: деление на ноль. В C++ деление на ноль вызывает неопределенное поведение — как в целочисленной арифметике, так и в арифметике с плавающей точкой. Каждый из трех наших компиляторов выдает предупреждения или сообщения об ошибках, если обнаруживает во время компиляции деление на ноль в целочисленной операции. Visual C++ также выдает сообщение об ошибке, если обнаруживает деление на ноль в операции с плавающей точкой. Целочисленное деление на ноль во время выполнения обычно приводит к сбою программы. Некоторые реализации C++ допускают деление числа с плавающей точкой на ноль и получают в результате «положительную бесконечность» или «отрицательную бесконечность», отображаемую как `inf` или `-inf` соответственно. Это относится и к трем нашим компиляторам.

Исходный код программы состоит из двух файлов:

- Файл `DivideByZeroException.h` (илл. 12.1) определяет пользовательский класс исключений, представляющий тип проблемы, которая может возникнуть в нашем примере.
- Файл `fig12_02.cpp` (илл. 12.2) определяет функцию `quotient` и функцию `main`, которая ее вызывает. Мы будем использовать их для объяснения процесса обработки исключений в системе управления.

12.2.1. Определение класса исключений для представления типа проблемы, которая может возникнуть

На илл. 12.1 определен класс `DivideByZeroException` — пользовательский класс исключений, которые наша программа выдает при обнаружении попыток деления на ноль. Мы определили его как производный класс стандартного библиотечного класса `runtime_error` (заголовок `<stdexcept>`). В типичном классе, производном от `runtime_error`, определяется только конструктор (строки 10–11), передающий строку сообщения об ошибке конструктору базового класса.

```
1 // Fig. 12.1: DivideByZeroException.h
2 // Определение класса DivideByZeroException
3 #include <stdexcept> // Заголовок stdexcept содержит класс runtime_error
4
5 // Объекты класса DivideByZeroException должны генерироваться
6 // функциями при обнаружении деления на ноль
7 class DivideByZeroException : public std::runtime_error {
8 public:
9    // Конструктор содержит сообщение об ошибке по умолчанию
10   DivideByZeroException()
11      : std::runtime_error{"attempted to divide by zero"} {}
12 };
```

Илл. 12.1 | Определение класса DivideByZeroException

Как правило, вы должны наследовать пользовательские классы исключений от классов стандартной библиотеки C++ и называть их в соответствии с обнаруженной проблемой. В C++ Core Guidelines разъясняется, что подобные пользовательские классы исключений труднее перепутать с исключениями, которые генерируют другие библиотеки, в том числе стандартная библиотека C++[1]. Подробнее о стандартных классах исключений мы расскажем в разделе 12.9.

12.2.2. Демонстрация обработки исключений

На илл. 12.2 показан пример **обертки (wrapping)** кода, который может вызвать исключение `DivideByZeroException`, и обработки этого исключения в случае его возникновения. Функция `quotient` получает два числа типа `double`, делит первое на второе и возвращает результат. Функция `quotient` рассматривает все попытки деления на ноль как ошибки. Если функция определяет, что второй аргумент равен нулю, она генерирует исключение `DivideByZeroException` (строка 13), чтобы сообщить о проблеме в точку вызова. Операнд в инструкции `throw` может быть любого **копируемого типа (copy-constructible type)**, а не только тех типов, которые прямо или косвенно являются производными от `exception`. Требуются именно копируемые типы, поскольку механизм обработки исключений копирует исключение во временный объект исключения[2].

```
1 // fig12_02.cpp
2 // Пример генерации исключения
3 // при попытке деления на ноль
4 #include <fmt/format.h>
5 #include <iostream>
6 #include "DivideByZeroException.h" // Класс DivideByZeroException
```

[1] C++ Core Guidelines, «E.14: Use Purpose-Designed User-Defined Types as Exceptions (not Builtin Types)». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-exception-types.

[2] «throw Expression». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/language/throw.

Окончание ⇨

```
 7
 8 // Выполняем деление, если знаменатель не равен нулю;
 9 // в противном случае создаем объект класса DivideByZeroException
10 double quotient(double numerator, double denominator) {
11    // Выдаем исключение DivideByZeroException при попытке деления на ноль
12    if (denominator == 0.0) {
13       throw DivideByZeroException{};
14    }
15
16    // Возвращаем результат деления
17    return numerator / denominator;
18 }
19
20 int main() {
21    int number1{0}; // Заданный пользователем числитель
22    int number2{0}; // Заданный пользователем знаменатель
23
24    std::cout << "Enter two integers (end-of-file to end): ";
25
26    // Пользователь вводит два целых числа для операции деления
27    while (std::cin >> number1 >> number2) {
28       // Блок try содержит код, который может вызвать исключение,
29       // и код, который не будет выполнен в случае исключения
30       try {
31          double result{quotient(number1, number2)};
32          std::cout << fmt::format("The quotient is: {}\n", result);
33       }
34       catch (const DivideByZeroException& divideByZeroException) {
35          std::cout << fmt::format("Exception occurred: {}\n",
36             divideByZeroException.what());
37       }
38
39       std::cout << "\nEnter two integers (end-of-file to end): ";
40    }
41
42    std::cout << '\n';
43 }
```

```
Enter two integers (end-of-file to end): 100 7
The quotient is: 14.2857

Enter two integers (end-of-file to end): 100 0
Exception occurred: attempted to divide by zero

Enter two integers (end-of-file to end): ^Z
```

Илл. 12.2 | Пример генерации исключения при попытке деления на ноль

Если пользователь указал корректный знаменатель, функция `quotient` возвращает результат деления. Если пользователь выбрал `0` в качестве знаменателя, функция `quotient` выдает исключение. В этом случае функция `main` обрабатывает исключение и просит пользователя ввести два новых значения, прежде чем снова

вызывает функцию `quotient`. Таким образом, программа продолжает выполняться даже после ввода неправильного значения, что делает ее надежнее. В выходных данных первые две строки показывают успешное вычисление, а следующие две показывают ошибку при попытке деления на ноль. После обсуждения этого кода мы рассмотрим пользовательские входные данные и потоки управления, которые генерируют выходные данные, показанные на илл. 12.3.

12.2.3. Включение кода в блок try

Программа предлагает пользователю ввести два целых числа и помещает их в условие цикла `while` (строка 27). В строке 31 введенные значения передаются в функцию `quotient` (строки 10–18), которая либо выполняет деление и возвращает результат, либо генерирует исключение, если знаменатель равен нулю. Строка 13 в функции `quotient` — это **точка выдачи**, или **точка выброса (throw point)**, исключения. Обработка исключений предназначена для ситуаций, когда функция, обнаружившая ошибку, не может выполнить свою задачу[1].

Блок `try` в строках 30–33 содержит две инструкции:

- вызов функции `quotient` (строка 31), которая может выдать исключение;
- инструкция вывода результата деления (строка 32).

Строка 32 выполняется только при успешном возврате результата функцией `quotient`.

12.2.4. Определение обработчика catch для DivideByZeroException

Сразу после блока `try` должен находиться как минимум один обработчик `catch` (строки 34–37). Параметр исключения должен быть объявлен как константная ссылка на обрабатываемый тип исключения — в данном случае `DivideByZeroException`. Это даст два ключевых преимущества:

- предотвращается копирование исключения;
- можно перехватывать исключения производных классов без среза (см. ниже).

Перехват исключения по ссылке позволяет избежать среза

Когда в блоке `try` появляется исключение, C++ выполняет первый обработчик `catch` с параметром того же типа, что и тип исключения, или с типом базового класса. Если обработчик `catch` базового класса перехватит объект исключения производного класса по значению, в параметр исключения будет скопирована только та часть объекта исключения производного класса, которая относится к базовому

[1] C++ Core Guidelines, «E.2: Throw an Exception to Signal That a Function Can't Perform Its Assigned Task». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-throw.

классу. Эта логическая ошибка, известная как **срез (slicing)**, возникает, когда объект производного класса копируется в объект базового класса или присваивается ему. Всегда перехватывайте исключения по ссылке, чтобы предотвратить срез.

Имя параметра в обработчике catch

Параметр исключения, который включает опциональное имя параметра (как в строке 34), позволяет обработчику `catch` взаимодействовать с перехваченным исключением. В строке 36 вызывается функция `what` для получения сообщения об ошибке.

Задачи, выполняемые обработчиками catch

Типичные задачи, выполняемые обработчиками `catch`:

- сообщить об ошибке пользователю;
- записать ошибку в файл, чтобы программисты могли изучить ее для отладки приложения;
- корректно завершить программу;
- осуществить альтернативные операции для выполнения задачи;
- перенаправить исключение в точку вызова;
- создать исключения другого типа.

Обработчик `catch` в нашем примере просто сообщает о попытке деления на ноль.

Распространенные ошибки при определении обработчиков catch

Начинающие программисты должны знать о некоторых ошибках, которые часто встречаются при обработке исключений:

- Размещение какого-либо кода между блоком `try` и его обработчиком `catch` или между любыми двумя его обработчиками `catch` — это синтаксическая ошибка.
- Каждый обработчик `catch` может иметь только один параметр. Список параметров, разделенных запятыми, — это синтаксическая ошибка.
- Перехват одного и того же типа исключения в нескольких обработчиках `catch` после одного блока `try` приведет к ошибке компиляции.
- Перехват исключения базового класса перед перехватом исключения производного класса — это логическая ошибка. Компиляторы обычно предупреждают вас в таких ситуациях.

12.2.5. Модель завершения обработки исключений

Если в блоке `try` не возникает исключений, управление программой продолжается с первой инструкции после последнего обработчика `catch`, следующего за данным блоком `try`. Если возникает исключение, блок `try` немедленно завершается и все

определенные в нем локальные переменные выходят из области видимости. В этом заключается преимущество обработки исключений. Все деструкторы локальных переменных в блоке `try` гарантированно выполняются, что позволяет программе избежать утечки ресурсов[1]. Далее программа ищет и выполняет первый соответствующий обработчик `catch`. Если поток управления достигает правой закрывающей фигурной скобки выполняемого обработчика `catch` (}), исключение считается обработанным. Любые локальные переменные в обработчике `catch`, включая параметр `catch`, выходят за пределы области видимости.

C++ использует модель завершения обработки исключений, в которой управление программой не может вернуться к точке выброса. Вместо этого управление возобновляется с первой инструкции (строка 39) после последнего обработчика `catch`, соответствующего блоку `try`.

Если в функции возникает исключение, которое не перехватывается в этой же области видимости, функция немедленно завершается. Программа пытается найти соответствующий блок `try` в вызывающей функции. Этот процесс, называемый размоткой стека, обсуждается в разделе 12.5.

12.2.6. Поток управления при вводе ненулевого знаменателя

Рассмотрим поток управления на илл. 12.3, когда пользователь вводит числитель `100` и знаменатель `7`. В строке 12 функция `quotient` определяет, что знаменатель не равен `0`, поэтому в строке 17 выполняется деление и результат (`14.2857`) возвращается в точку вызова (строка 31). Программа продолжается последовательно со строки 31, поэтому в строке 32 отображается результат деления, и поток управления достигает конечной фигурной скобки блока `try`. В этом случае блок `try` успешно завершил выполнение, поэтому программа пропускает обработчик `catch` (строки 34–37) и продолжается со строки 39.

12.2.7. Поток управления при вводе нулевого знаменателя

Теперь рассмотрим поток управления при вводе пользователем числителя `100` и знаменателя `0`. В строке 12 функция `quotient` определяет, что знаменатель равен `0`, поэтому в строке 13 используется ключевое слово `throw` для генерации объекта `DivideByZeroException`. Это вызывает конструктор класса `DivideByZeroException` для инициализации объекта исключения. Конструктор нашего класса не имеет параметров. Если у конструктора исключения есть один или несколько параметров, вы должны передать ему соответствующие аргументы при создании объекта, например:

```
throw out_of_range{"Index out of range"};
```

[1] «Technical Report on C++ Performance», 15 февраля 2006 г. Был доступен: 15 сентября 2023 г. http://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf (страница 34).

Когда выдается исключение, функция `quotient` немедленно завершается, операция деления не выполняется. Если вы явно выдаете исключение в собственном коде, как правило, вы это делаете до появления ошибки. Но это не всегда возможно. Например, функция может вызвать другую функцию, которая обнаружит ошибку и выдаст исключение.

Мы поместили вызов функции `quotient` (строка 31) в блок `try`, поэтому поток управления переходит в первый соответствующий обработчик `catch` (строки 34–37), следующий непосредственно за блоком `try`. В этой программе обработчик `catch` перехватывает исключение `DivideByZeroException` (тип, созданный функцией `quotient`) и выводит сообщение об ошибке, возвращаемое функцией `what`. Если разным типам ошибок, возникающих во время выполнения программы, соответствуют исключения с подходящими именами, ваш код становится более понятным.

12.3. Гарантии безопасности исключений и noexcept

Программисты клиентского кода должны понимать суть происходящего при использовании вашего кода. Есть ли такая возможность для исключений? Если возможность есть, то как будет себя вести программа в случае генерации исключения? Когда вы разрабатываете свой код, подумайте, какие **гарантии безопасности исключений (exception safety guarantees)** вы можете предоставить[1,2]:

- Отсутствие гарантии: если возникает исключение, программа может немедленно завершиться с утечкой ресурсов, таких как динамически выделенная память.
- Базовая гарантия: если возникает исключение, утечка ресурсов не происходит, объекты не уничтожаются, но их состояния могут измениться. Как правило, код, который может генерировать исключения, должен обеспечивать как минимум базовую гарантию.
- Строгая гарантия: если возникает исключение, состояния объектов остаются неизмененными или, если объекты были изменены, они возвращаются в те же состояния, которые были до операции, вызвавшей исключение. Мы реализовали строгую гарантию для оператора копирующего присваивания класса `myArray` в главе 11 с помощью идиомы копирования и обмена. Оператор сначала копирует аргумент — это операция, которая может завершиться ошибкой при выделении ресурсов. В этом примере присваивание или завершается ошибкой без изменения исходного объекта, или завершается успешно.
- Гарантия отсутствия исключений: операция не генерирует исключения. Например, в главе 11 перемещающий конструктор класса `myArray`, оператор пе-

[1] Клаус Игльбергер (Klaus Iglberger), «Back to Basics: Exceptions», 5 октября 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=0ojB8c0xUd8.

[2] «Exceptions». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/language/exceptions.

ремещающего присваивания и дружественная функция swap были объявлены как noexcept. Они работают с существующими ресурсами, не создают новые ресурсы, поэтому не могут столкнуться с ошибкой. Только функции, не имеющие возможности завершить выполнение ошибкой, должны быть объявлены как noexcept. Если в функции, объявленной как noexcept, появляется исключение, программа немедленно завершается.

12.4. Повторная генерация исключений

Иногда вы можете:

- перехватить исключение;
- частично его обработать;
- уведомить вызывающий код, повторно выдав исключение.

Этот процесс означает, что исключение еще не обработано. Инструкция

```
throw;
```

выполняемая в обработчике catch, возвращает текущее исключение в следующий блок try. Затем обработчик catch, расположенный после этого блока try, пытается обработать исключение. Выполнение такой инструкции вне обработчика catch немедленно завершает программу.

Если обработчик catch содержит имя параметра для перехватываемого исключения, например ex, не выдавайте исключение повторно с помощью инструкции

```
throw ex;
```

Это приведет к ненужному копированию исходного объекта исключения. Это также может быть логической ошибкой. Если тип исключения совпадает с типом базового класса исключения, повторная генерация приведет к размотке стека.

Варианты повторной генерации исключений

Существуют различные варианты повторной генерации исключений:

- Вам надо записать исключение в журнал для дальнейшей отладки приложения. В этом случае вы перехватываете исключение, регистрируете его и повторно генерируете исключение для дальнейшей обработки в программе. Мы обсуждаем ведение журнала исключений в разделе 12.12.
- Вам надо создать исключение другого типа, более специфичное для вашей библиотеки или для вашего приложения. В этом случае вы можете **обернуть** (**wrap**) исходное исключение в новое, используя класс nested_exception (C++11)[1].

[1] «std::nested_exception». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/error/nested_exception.

- Вы выполняете частичную обработку исключения, например освобождаете ресурс, и повторно выдаете исключение для дальнейшей обработки в обработчике `catch` блока `try`.
- В некоторых случаях исключения генерируются повторно неявным образом, например в функциональных блоках `try` конструкторов и деструкторов. Мы обсуждаем это в разделе 12.7.2.

Пример повторной генерации исключения

На илл. 12.3 показан пример повторной генерации исключения. В блоке `try` функции `main` (строки 24–28) в строке 26 вызывается функция `throwException` (строки 7–20), содержащая блок `try` (строки 9–12), из которого инструкция `throw` в строке 11 выдает объект исключения. Обработчик `catch` (строки 13–17) перехватывает исключение, выводит сообщение об ошибке (строки 14–15) и повторно генерирует исключение (строка 16). Эта инструкция немедленно возвращает управление в строку 26 блока `try` функции `main`. Этот блок `try` завершается (поэтому инструкция в строке 27 не выполняется), а обработчик `catch` в функции `main` (строки 29–31) перехватывает исключение и выводит сообщение об ошибке (строка 30). Мы не используем параметры исключения в обработчиках `catch` этого примера, поэтому опускаем имена параметров исключения и указываем только тип исключения для перехвата (строки 13 и 29).

```
1 // fig12_03.cpp
2 // Повторная генерация исключения
3 #include <iostream>
4 #include <exception>
5
6 // Выдаем, перехватываем и снова выдаем исключение
7 void throwException() {
8    // Выдаем исключение и немедленно его перехватываем
9    try {
10      std::cout << " Function throwException throws an exception\n";
11      throw std::exception{}; // Точка выдачи исключения
12   }
13   catch (const std::exception&) { // Обработчик исключения
14      std::cout << " Exception handled in function throwException"
15         << "\n Function throwException rethrows exception";
16      throw; // Снова выдаем исключение для дальнейшей обработки
17   }
18
19   std::cout << "This should not print\n";
20 }
21
22 int main() {
23   // Вызываем функцию throwException
24   try {
25      std::cout << "main invokes function throwException\n";
26      throwException();
```

```
27       std::cout << "This should not print\n";
28    }
29    catch (const std::exception&) { // Обработчик исключения
30       std::cout << "\n\nException handled in main\n";
31    }
32
33    std::cout << "Program control continues after catch in main\n";
34 }
```

```
main invokes function throwException
  Function throwException throws an exception
  Exception handled in function throwException
  Function throwException rethrows exception

Exception handled in main
Program control continues after catch in main
```

Илл. 12.3 | Повторная генерация исключения

12.5. Размотка стека и необработанные исключения

Когда вы не перехватываете исключение в определенной области видимости, функция вызывает размотку стека в попытке перехватить исключение во внешней области видимости, то есть в блоке `catch` внешнего блока `try`. Вы можете делать блоки `try` вложенными, и в этом случае следующий внешний блок может находиться в той же функции. Когда блоки `try` не вложены друг в друга, происходит размотка стека. Функция, в которой исключение не было перехвачено, завершается, и ее локальные переменные выходят из области видимости. Во время размотки стека управление возвращается к инструкции, вызвавшей эту функцию. Если эта инструкция находится в блоке `try`, блок завершается и предпринимается попытка перехватить исключение. Если эта инструкция не помещена в блок `try` или исключение не было обработано, размотка стека продолжается. Это одна из причин, по которой вы не должны оборачивать инструкции `try...catch` вокруг каждого вызова функции, где возможна генерация исключения[1]. Иногда лучше поручить решение проблемы внешней функции из цепочки вызовов. На илл. 12.4 показан пример размотки стека.

```
1 // fig12_04.cpp
2 // Демонстрация размотки стека
3 #include <iostream>
4 #include <stdexcept>
5
6 // Функция function3 генерирует исключение
```

[1] C++ Core Guidelines, «E.17: Don't Try to Catch Every Exception in Every Function». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-not-always.

Окончание ⇨

```
 7 void function3() {
 8    std::cout << "In function 3\n";
 9
10    // Нет блока try, размотка стека возвращает управление функции function2
11    throw std::runtime_error{"runtime error in function3"};
12 }
13
14 // Функция function2 вызывает function3
15 void function2() {
16    std::cout << "function3 is called inside function2\n";
17    function3(); // Размотка стека возвращает управление функции function1
18 }
19
20 // Функция function1 вызывает function2
21 void function1() {
22    std::cout << "function2 is called inside function1\n";
23    function2(); // Размотка стека возвращает управление функции main
24 }
25
26 // Демонстрация размотки стека
27 int main() {
28    // Вызываем function1
29    try {
30       std::cout << "function1 is called inside main\n";
31       function1(); // Функция function1 выдает исключение runtime_error
32    }
33    catch (const std::runtime_error& error) { // Обрабатываем исключение
34       std::cout << "Exception occurred: " << error.what()
35          << "\nException handled in main\n";
36    }
37 }
```

```
function1 is called inside main
function2 is called inside function1
function3 is called inside function2
In function 3
Exception occurred: runtime_error in function3
Exception handled in main
```

Илл. 12.4 | Демонстрация размотки стека

В функции `main`, в блоке `try` (строка 31) вызывается функция `function1` (строки 21–24), вызывающая функцию `function2` (строки 15–18), которая, в свою очередь, вызывает `function3` (строки 7–12). Функция `function3` в строке 11 генерирует объект `runtime_error` — это точка выброса. Далее управление происходит следующим образом:

- Ни один блок `try` не обрабатывает строку 11, поэтому начинается размотка стека. Вызов `function3` завершается, возвращая управление в строку 17 в `function2`.

- Ни один блок `try` не обрабатывает строку 17, поэтому размотка стека продолжается. Вызов `function2` завершается, возвращая управление в строку 23 в `function1`.
- Ни один блок `try` не обрабатывает строку 23, поэтому размотка стека продолжается. Вызов `function1` завершает работу и возвращает управление в строку 31 в `main`.
- Блок `try` в строках 29–32 обрабатывает строку 31, поэтому первый соответствующий обработчик `catch` блока `try` (строки 33–36) перехватывает и обрабатывает исключение, выводя сообщение о возникшей ошибке.

Неперехваченные исключения

Если исключение не будет перехвачено во время размотки стека, C++ вызовет функцию `terminate` стандартной библиотеки, которая вызовет функцию `abort` для завершения программы. Для демонстрации этого механизма мы удалили обработку `try...catch` из функции `main` в файле `fig12_04.cpp`, сохранив только строки 30–31 из блока `try` на илл. 12.4. Эта модифицированная версия программы `fig12_04modified.cpp` находится в папке примера `fig12_04`. Когда вы запустите модифицированную версию и исключение не будет перехвачено в функции `main`, программа завершится. Ниже показаны выходные данные программы в GNU C++. Обратите внимание, что в выходных данных упоминается вызов функции `terminate`:

```
function1 is called inside main
function2 is called inside function1
function3 is called inside function2
In function 3
terminate called after throwing an instance of 'std::runtime_error'
  what(): runtime_error in function3
Aborted (core dumped)
```

12.6. Когда надо обрабатывать исключения

Обработка исключений предназначена для коррекции редких **синхронных ошибок (synchronous errors)**, которые могут возникать во время выполнения инструкций, даже если ваш код написан правильно[1]. Примеры таких ошибок:

- попытка доступа к временно недоступному веб-сервису;
- попытка чтения из несуществующего файла;
- попытка получить доступ к файлу без прав доступа;
- сбой выделения динамической памяти и т. д.

[1] «Modern C++ Best Practices for Exceptions and Error Handling». Был доступен: 15 сентября 2023 г. https://docs.microsoft.com/en-us/cpp/cpp/errors-and-exception-handling-modern-cpp.

Обработка исключений не предназначена для обработки **асинхронных событий (asynchronous events)**, происходящих одновременно с потоком управления программы и независимых от него. Примеры таких событий — это завершение ввода-вывода, поступление сетевых сообщений, щелчки мыши и нажатия клавиш.

Сложные приложения обычно состоят из предопределенных программных компонентов (например, классов стандартной библиотеки) и компонентов, специфичных для конкретного приложения, использующих предопределенные компоненты. Когда предопределенный компонент сталкивается с проблемой, он должен сообщить о проблеме специфичному компоненту. Предопределенный компонент не может знать, как разные приложения решают проблему. Иногда о проблеме надо сообщить функции, находящейся в начале цепочки вызовов, генерирующих исключение.

Обработка исключений обеспечивает единый метод обработки проблем. В крупных проектах это помогает программистам понимать сторонний код, обрабатывающий исключения. Это также позволяет предопределенным программным компонентам (например, классам стандартной библиотеки) сообщать о проблемах специфичным компонентам. Вы должны включить вашу стратегию обработки исключений в свою систему на этапе ее проектирования[1] — сделать это после внедрения системы будет значительно труднее.

Когда не надо обрабатывать исключения

На сайте isocpp.org, в разделе часто задаваемых вопросов об обработке исключений, перечислены несколько сценариев, в которых программист не должен использовать исключения[2]. В частности:

- когда сбои ожидаются с большой вероятностью, например при преобразовании неформатированных строк в числовые значения, если строки могут иметь неправильный формат;
- когда установлены строгие требования к быстродействию, из-за которых дополнительная нагрузка при генерации исключений и размотке стека неприемлема, например в бортовых системах реального времени самолетов Joint Strike Fighter, для которых существует отдельный стандарт C++[3];
- для исправления допущенных программистом ошибок, которых не должно быть в коде, таких как доступ к элементам массива, находящимся вне диапазона, разыменование нулевого указателя или деление на ноль[4]. Интересно,

---

[1] C++ Core Guidelines, «E.1: Develop an Error-Handling Strategy Early in a Design». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-design.

[2] «What Shouldn't I Use Exceptions For?». Был доступен: 15 сентября 2023 г. https://isocpp.org/wiki/faq/exceptions#why-not-exceptions.

[3] «Joint Strike Fighter Air Vehicle C++ Coding Standards». Декабрь 2005 г. Был доступен: 15 сентября 2023 г. https://www.stroustrup.com/JSF-AV-rules.pdf.

[4] Мы использовали деление на ноль только для демонстрации потока управления при обработке исключения.

что стандарт C++ включает класс `out_of_range`, а различные классы стандартной библиотеки, такие как `array` и `vector`, содержат функции, генерирующие исключения `out_of_range`.

Обработка исключений может увеличить размер исполняемого файла вашей программы[1], что бывает неприемлемо на устройствах с малым объемом памяти, таких как встроенные системы. Впрочем, в разделе часто задаваемых вопросов по обработке исключений на сайте isocpp.org отмечается, что «обработка исключений вам почти ничего не стоит, пока вы не выдаете исключение. В некоторых реализациях — совсем ничего. Все расходы возникают, когда вы выдаете исключение, то есть программа с исключениями в норме выполняется быстрее, чем программа с кодами возврата ошибок и тестами. Вы платите цену только при встрече с ошибкой»[2]. Проблемы быстродействия при обработке исключений подробно рассмотрены в документе «Technical Report on C++ Performance», в разделе 5.4[3].

Функции, которые встречаются с распространенными ошибками, как правило, должны возвращать `nullptr`, `0` или другие соответствующие значения (например, `bool`) вместо генерации исключений. Программа, вызывающая такую функцию, может проверить возвращаемое значение, чтобы определить, был ли вызов функции успешным или нет. Херб Саттер — организатор Комитета ISO по стандартам C++ (WG21) и разработчик программного обеспечения в Microsoft — поясняет: «Баги в программах не относятся к тем ошибкам, которые можно исправить во время выполнения, поэтому их не нужно сопровождать исключениями или соответствующими кодами ошибок». Далее он говорит, что уже идет процесс изменения стандартных библиотек C++, чтобы они не генерировали исключения для подобных ошибок[4]. В разделе 12.13, посвященном контрактам, вы увидите, что новые возможности контрактов (первоначально запланированных для C++20, но отложенных как минимум до C++23) помогут снизить потребность в исключениях стандартной библиотеки.

В C++ Core Guidelines поясняется, что большое количество инструкций `try...catch` в вашем коде может быть признаком слишком плохого управления ресурсами[5]. Чтобы избежать необходимости в инструкциях `try...catch`, в C++ Core Guidelines рекомендуется проектировать классы с использованием концепции RAII («получение ресурсов есть инициализация», глава 11). Напомним, что в этой концепции конструктор объекта создает ресурсы, а деструктор объекта их освобождает. Таким

---

[1] Вишал Човатия (Vishal Chovatiya), «C++ Exception Handling Best Practices: 7 Things To Know», 3 ноября 2019 г. Был доступен: 15 сентября 2023 г. http://www.vishalchovatiya.com/7-best-practices-forexception-handling-in-cpp-with-example/.

[2] «Why Use Exceptions?». Был доступен: 15 сентября 2023 г. https://isocpp.org/wiki/faq/exceptions#why-exceptions.

[3] «Technical Report on C++ Performance», 15 февраля 2006 г. Был доступен: 15 сентября 2023 г. http://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf.

[4] Херб Саттер (Herb Sutter), «Zero-Overhead Deterministic Exceptions: Throwing Values», 4 августа 2019 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p0709R4.

[5] C++ Core Guidelines, «E.18: Minimize the Use of Explicit try/catch». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-catch.

образом, если объект выходит из области видимости по какой-либо причине, включая генерацию исключения, ресурсы объекта будут освобождены.

12.6.1. Макрос assert

При реализации и отладке программ иногда требуется указывать условия, которые должны быть истинными в определенной точке функции. Эти условия, называемые **утверждениями (assertions)**, помогают обеспечить корректную работу программы путем перехвата потенциальных ошибок и выявления возможных логических ошибок во время разработки. Утверждение — это выполняемая во время работы программы проверка какого-либо условия, которое всегда должно быть истинным, если ваш код написан правильно. Если условие оказалось ложным, программа немедленно завершается, выводя сообщение об ошибке, включающее имя файла, номер строки, в которой возникла проблема, и условие, при котором произошел сбой. Утверждения реализуются с помощью макроса `assert` из заголовка `<cassert>`[1], например:

```
assert(условие);
```

Утверждения — это прежде всего вспомогательный инструмент[2] программиста, предназначенный для выявления ошибок в коде, которые необходимо исправить. Например, вы можете использовать утверждение в функции, обрабатывающей массивы, чтобы убедиться, что индексы каждого массива больше `0` и меньше длины массива. После завершения отладки все утверждения можно отключить, добавив соответствующую директиву препроцессора перед `#include <cassert>`:

```
#define NDEBUG
```

Как правило, компиляторы предоставляют параметр, который позволяет отключать утверждения без изменения кода. Например, в g++ для этого служит опциональный параметр командной строки `-DNDEBUG`.

12.6.2. Немедленное завершение

В C++ Core Guidelines предлагается: «Если вы не можете выдавать исключения, рассмотрите возможность немедленного завершения программы»[3]. **Немедленное завершение (fail-fast)** — это такой стиль разработки, при котором вы не перехватываете и не обрабатываете исключения, оставляя программу в состоянии возможности дальнейших сбоев, а немедленно ее завершаете[4]. Это кажется не-

---

[1] «assert». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/error/assert.

[2] «Modern C++ Best Practices For Exceptions and Error Handling», 24 августа 2020 г. Был доступен: 15 сентября 2023 г. https://docs.microsoft.com/en-us/cpp/cpp/errors-and-exception-handling-modern-cpp.

[3] C++ Core Guidelines, «E.26: If You Can't Throw Exceptions, Consider Failing Fast». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-no-throw-crash.

[4] «Fail-fast». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Fail-fast.

логичным, но идея заключается в том, что немедленное завершение помогает создавать более надежное программное обеспечение, находя и исправляя ошибки в процессе разработки. При таком подходе можно рассчитывать, что в готовом продукте останется меньше ошибок[1].

12.7. Конструкторы, деструкторы и обработка исключений

Есть целый ряд неочевидных проблем, связанных с исключениями в контексте конструкторов и деструкторов. В этом разделе мы расскажем:

- почему конструкторы должны генерировать исключения при возникновении ошибок;
- как перехватывать исключения, возникающие при инициализации элементов данных в конструкторах;
- почему деструкторы не должны генерировать исключения.

12.7.1. Генерация исключений в конструкторах

Для начала рассмотрим проблему, о которой мы упоминали, но еще не решали. Что происходит, когда в конструкторе появляются ошибки? Например, как должен реагировать конструктор объекта, когда он получает недопустимые данные? Поскольку конструктор не может вернуть сообщение об ошибке, мы должны каким-то образом указать, что объект не был инициализирован должным образом. Один из способов решения — вернуть неправильно инициализированный объект и надеяться, что кто-то, использующий его, проведет соответствующие тесты и обнаружит неправильное состояние. Другой способ — создать переменную вне конструктора, например глобальную переменную для обозначения ошибок, но это считается плохой практикой разработки программного обеспечения. Предпочтительной альтернативой будет требование к конструктору генерировать исключения, содержащие информацию об ошибке. Однако не следует выдавать исключения из конструктора глобального объекта. Такие исключения не могут быть перехвачены, потому что они создаются до выполнения функции `main`.

Конструктор должен генерировать исключение, если при инициализации объекта возникает проблема. Если конструктор управляет динамической памятью без умных указателей, он должен освободить эту память перед созданием исключения, чтобы предотвратить утечку памяти. Деструктор не будет вызываться для освобождения ресурсов, хотя будут вызываться деструкторы для уже созданных элементов данных. Всегда используйте умные указатели для управления динамической памятью.

---

[1] Джим Шор (Jim Shore), «Fail Fast [Software Debugging]». IEEE Software 21, № 5 (сентябрь-октябрь 2004 г.): 21–25. Под ред. Мартина Фаулера (Martin Fowler). https://martinfowler.com/ieeeSoftware/failFast.pdf.

12.7.2. Перехват исключений в конструкторах с помощью функциональных блоков try

Напомним, что инициализаторы базового класса и инициализаторы элементов данных выполняются перед телом конструктора. Поэтому если вы хотите перехватывать исключения, создаваемые этими инициализаторами, вы не можете просто обернуть инструкции тела конструктора в блок `try`. Вместо этого вы должны использовать функциональный блок `try`, как показано на илл. 12.5.

```
// fig12_05.cpp
// Демонстрация функционального блока try
#include <fmt/format.h>
#include <iostream>
#include <limits>
#include <stdexcept>

// Класс Integer намеренно выдает исключение из своего конструктора
class Integer {
public:
   explicit Integer(int i) : value{i} {
      std::cout << fmt::format("Integer constructor: {}\n", value)
         << "Purposely throwing exception from Integer constructor\n";
      throw std::runtime_error("Integer constructor failed");
   }
private:
   int value{};
};

class ResourceManager {
public:
   ResourceManager(int i) try : myInteger(i) {
      std::cout << "ResourceManager constructor called\n";
   }
   catch (const std::runtime_error& ex) {
      std::cout << fmt::format(
         "Exception while constructing ResourceManager: ", ex.what())
         << "\nAutomatically rethrowing the exception\n";
   }
private:
   Integer myInteger;
};

int main() {
   try {
      const ResourceManager resource{7};
   }
   catch (const std::runtime_error& ex) {
      std::cout << fmt::format("Rethrown exception caught in main: {}\n",
         ex.what());
   }
}
```

```
Integer constructor: 7
Purposely throwing exception from Integer constructor
Exception while constructing ResourceManager: Integer constructor failed
Automatically rethrowing the exception
Rethrown exception caught in main: Integer constructor failed
```

Илл. 12.5 | Функциональный блок try

Чтобы продемонстрировать работу функционального блока `try`, класс `Integer` (строки 9–18) имитирует невозможность получить ресурс, намеренно генерируя исключение (строка 14) в своем конструкторе. Класс `ResourceManager` (строки 20–32) содержит объект класса `Integer` (строка 31), который будет инициализирован в списке инициализаторов конструктора `ResourceManager`.

В конструкторе вы определяете функциональный блок `try`, помещая ключевое слово `try` после списка параметров конструктора и перед двоеточием (`:`), которое предваряет список инициализаторов (строка 22). За списком инициализаторов следует тело конструктора (строки 22–24). Любые исключения, возникающие в списке инициализаторов или в теле конструктора, могут обрабатываться блоками `catch`, следующими за телом конструктора, — в нашем примере это блок `catch` в строках 25–29. Поток управления в этом примере выглядит следующим образом:

- В строке 36 в функции `main` создается объект класса `ResourceManager`, при этом вызывается его конструктор.
- В этом конструкторе, в строке 22, вызывается конструктор класса `Integer` (строки 11–15) для инициализации объекта `myInteger` и для вывода первых двух строк выходных данных. В строке 14 намеренно генерируется исключение, чтобы мы могли продемонстрировать функциональный блок `try` конструктора `ResourceManager`. Исключение завершает работу конструктора класса `Integer` и возвращается инициализатору в строку 22, которая находится в функциональном блоке `try` конструктора `ResourceManager`.
- Функциональный блок `try`, также включающий тело конструктора, завершается.
- Обработчик `catch` конструктора `ResourceManager` в строках 25–29 перехватывает исключение и выводит следующие две строки выходных данных.
- Основная цель функционального блока `try` конструктора — дать вам возможность выполнить первоначальную обработку исключения, например запротоколировать исключение в журнале или выдать другое, более подходящее для вашего кода исключение. Ваш объект не может быть создан, поэтому каждый обработчик `catch`, следующий за функциональным блоком `try`, должен либо генерировать новое исключение, либо повторно генерировать существующее — явно или неявно[1]. Наш обработчик `catch` не содержит явную

[1] «Function-try-Block». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/language/function-try-block.

инструкцию `throw`, поэтому он неявно генерирует исключение повторно. Это завершает работу конструктора `ResourceManager` и возвращает исключение обратно в строку 36 в функции `main`.

- Строка 36 находится в блоке `try`, и этот блок тоже завершается, а обработчик `catch` в строках 38–41 обрабатывает исключение, выводя последнюю строку выходных данных.

Функциональные блоки `try` также можно использовать с функциями:

```
void myFunction() try {
   // Тело функции
}
catch (const ExceptionType& ex) {
   // Обработка исключения
}
```

Однако в обычных функциях подобное использование функционального блока `try` не дает дополнительных преимуществ по сравнению с традиционным размещением последовательности инструкций `try...catch` непосредственно в теле функции:

```
void myFunction() {
   try {
      // Тело функции
   }
   catch (const ExceptionType& ex) {
      // Обработка исключения
   }
}
```

12.7.3. Исключения, деструкторы и noexcept(false)

Начиная с C++11, компилятор неявно объявляет все деструкторы как `noexcept`, за исключением двух случаев:

- если вы явно объявили деструктор как `noexcept(false)`;
- если прямой или косвенный деструктор базового класса объявлен как `noexcept(false)`.

Если ваш деструктор вызывает функции, которые могут выдать исключения, вы должны перехватить и обработать их, даже если обработка заключается только в записи ошибки в журнал и контролируемом завершении программы[1]. Во время уничтожения объекта производного класса, если деструктор базового класса может выдать исключение, вы можете использовать функциональный блок `try` для деструктора производного класса, чтобы гарантировать возможность перехвата исключения.

[1] C++ Core Guidelines. «C.36: A Destructor Must Not Fail». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-dtor-fail.

Если при создании объекта генерируется исключение, деструктор может быть вызван:

- для объектов и элементов данных, созданных до генерации исключения, если объект не был создан полностью;
- для созданной части объектов массива, если исключение было сгенерировано во время создания массива объектов.

Размотка стека будет гарантированно завершена, когда начнет выполняться обработчик `catch`. Если вызванный при размотке стека деструктор выдает исключение, программа завершается. На сайте **isocpp.org**, в разделе часто задаваемых вопросов[1], это объясняется тем, что C++ не знает, следует ли:

- продолжить обработку исключения, которое привело к размотке стека;
- обработать новое исключение, генерируемое деструктором.

12.8. Обработка исключений оператора new

В разделе 11.4 мы рассмотрели выделение динамической памяти оператором `new`, а в разделе 11.5 показали современный способ управления памятью в C++ в соответствии с идиомой RAII («получение ресурсов есть инициализация»): с помощью шаблона класса `unique_ptr` и шаблона функции `make_unique`, которая реализует работу с оператором `new` «под капотом». Если оператору `new` не удается выделить запрашиваемую память, он выдает исключение `bad_alloc` (определенное в заголовке `<new>`).

В этом разделе на илл. 12.6–12.7 показаны два примера с исключениями оператора `new`. В каждом из них осуществляется попытка получить большие объемы динамической памяти:

- в первом примере демонстрируется исключение `bad_alloc`;
- во втором примере мы вызываем функцию `set_new_handler`, чтобы указать ту функцию, которая будет вызываться при сбое оператора `new`, — этот метод применялся в устаревшем коде C++, написанном до начала поддержки компиляторами исключения `bad_alloc`.

Когда исключение генерируется конструктором объекта, создаваемого в инструкции с оператором `new`, выделенная для этого объекта динамическая память освобождается.

Не создавайте исключения при работе с простыми указателями на динамически выделяемую память

В разделе 11.5 показано, что указатель `unique_ptr` гарантирует правильное освобождение динамически выделяемой памяти независимо от выхода указателя

[1] «How can I handle a destructor that fails?» Был доступен: 15 сентября 2023 г. https://isocpp.org/wiki/faq/exceptions#dtors-shouldnt-throw.

из области видимости при нормальном потоке управления или при генерации исключения. При управлении динамически выделяемой памятью в старом стиле, с помощью простых указателей (что встречается в устаревшем коде C++), не допускайте неперехваченных исключений до освобождения памяти[1]. Например, предположим, что функция содержит такой ряд инструкций:

```
int* ptr{new int[100]}; // Получаем динамически выделяемую память
processArray(ptr); // Предположим, что эта функция может выдать исключение
delete[] ptr; // Возвращаем память операционной системе
   // ...
```

Утечка памяти может произойти, если функция `processArray` выдаст исключение, — код не выполнит инструкцию `delete[]`. Чтобы предотвратить утечку памяти, вам нужно перехватить исключение и освободить память раньше, чем исключение будет передано в точку вызова. Поэтому вы должны управлять памятью с помощью умных указателей `unique_ptr`, как описано в главе 11.

12.8.1. Генерация исключения bad_alloc оператора new

На илл. 12.6 показано исключение `bad_alloc`, которое выдает оператор `new`, когда ему не удается выделить память. Инструкция `for` (строки 15–19), заключенная в блок `try`, перебирает массив объектов `unique_ptr` с именем `items` и выделяет память каждому элементу массива, состоящему из `500 000 000` элементов типа `double`. Наш основной тестовый компьютер имеет 32 Гбайта оперативной памяти и 8 Тбайт дискового пространства. Нам пришлось выделить память для огромного количества элементов, чтобы вызвать сбой выделения динамической памяти. Тогда мы решили использовать отдельный тестовый компьютер с 16 Гбайт оперативной памяти и 256 Гбайт дискового пространства, чтобы получить нужные выходные данные. Но все равно нам пришлось выделить память для пяти миллиардов элементов `double`, чтобы вызвать сбой. Возможно, на вашем компьютере вам, наоборот, потребуется уменьшить в коде число `500 000 000`. Если оператор `new` генерирует исключение `bad_alloc` во время вызова функции `make_unique` и выдает исключение, цикл завершается. Программа продолжает выполнение в строке 21, где обработчик `catch` перехватывает и обрабатывает исключение. В строках 22–23 выводится сообщение `"Exception occurred:"`, а затем сообщение из функции `what`. Как правило, оно определяется реализацией, например `"bad allocation"` или `"std::bad_alloc"`. Выходные данные показывают, что программа выполнила только десять итераций цикла, прежде чем оператор `new` выдал исключение `bad_alloc`. Ваши выходные данные могут отличаться в зависимости от вашего компилятора, объема физической памяти и дискового пространства, доступного для виртуальной памяти в вашей системе.

[1] C++ Core Guidelines, «E.13: Never Throw While Being the Direct Owner of an Object». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-never-throw.

```
// fig12_06.cpp
// Демонстрация стандартного исключения bad_alloc,
// которое оператор new выдает при недостатке памяти
#include <array>
#include <fmt/format.h>
#include <iostream>
#include <memory>
#include <new> // Определение класса bad_alloc

int main() {
   std::array<std::unique_ptr<double[]>, 1000> items{};

   // Направляем каждый unique_ptr на большой блок памяти
   try {
      for (int i{0}; auto& item : items) {
         item = std::make_unique<double[]>(500'000'000);
         std::cout << fmt::format(
            "items[{}] points to 500 000 000 doubles\n", i++);
      }
   }
   catch (const std::bad_alloc& memoryAllocationException) {
      std::cerr << fmt::format("Exception occurred: {}\n",
         memoryAllocationException.what());
   }
}
```

```
items[0] points to 500 000 000 doubles
items[1] points to 500 000 000 doubles
items[2] points to 500 000 000 doubles
items[3] points to 500 000 000 doubles
items[4] points to 500 000 000 doubles
items[5] points to 500 000 000 doubles
items[6] points to 500 000 000 doubles
items[7] points to 500 000 000 doubles
items[8] points to 500 000 000 doubles
items[9] points to 500 000 000 doubles
Exception occurred: bad allocation
```

Илл. 12.6 | Оператор new генерирует исключение bad_alloc

12.8.2. Оператор new, возвращающий nullptr при сбое

Как правило, вы должны использовать версию оператора new, генерирующую исключение bad_alloc. Однако в стандарте C++ указано, что вы также можете использовать устаревшую версию оператора new, которая возвращает nullptr при сбое. Для этой цели в заголовке <new> определен объект std::nothrow (типа nothrow_t), используемый таким образом:

```
std::unique_ptr<double[]> ptr{
   new(std::nothrow) double[500'000'000]};
```

Здесь вы не можете использовать функцию make_unique, потому что она использует версию оператора new по умолчанию.

12.8.3. Обработка исключений оператора new с помощью функции set_new_handler

В устаревшем коде C++ вы можете встретиться с функцией set_new_handler (заголовок <new>), предназначенной для обработки исключений оператора new. Она принимает в качестве аргумента:

- указатель на функцию, которая не принимает аргументы и возвращает void;

или

- лямбду, которая не принимает аргументы и возвращает void.

Эта функция или лямбда вызывается, когда оператор new генерирует исключение. Как только функция set_new_handler регистрирует обработчик оператора new в программе, оператор new перестает выдавать исключение bad_alloc в случае сбоя. Вместо этого он делегирует обработку ошибок функции-обработчику исключений оператора new.

Функция-обработчик исключений оператора new должна выполнить одну из следующих задач:

1. Расширить доступную для выделения память — освободить другую область динамически выделенной памяти или уведомить пользователя о необходимости закрыть другие приложения, затем попробовать выделить память еще раз.
2. Выдать исключение bad_alloc (или исключение производного класса от bad_alloc).
3. Вызвать функцию abort или exit (обе из заголовка <cstdlib>) для завершения программы. Функция abort немедленно завершает программу, а функция exit перед завершением программы вызывает деструкторы глобальных объектов и локальных статических объектов. При вызове любой из этих функций нестатические локальные объекты не уничтожаются.

На илл. 12.7 показана работа функции set_new_handler (строка 21). Функция customNewHandler (строки 11–14) выводит сообщение об ошибке (строка 12), затем вызывает функцию exit (строка 13) для завершения работы программы. Константа EXIT_FAILURE определена в заголовке <cstdlib>. Выходные данные показывают, что цикл повторился девять раз, прежде чем оператор new не смог выделить память и вызвал функцию customNewHandler. Ваши выходные данные могут отличаться в зависимости от вашего компилятора, объема физической памяти и дискового пространства, доступного для виртуальной памяти в вашей системе.

```
// fig12_07.cpp
// Демонстрация функции set_new_handler
#include <array>
#include <cstdlib>
#include <fmt/format.h>
#include <iostream>
#include <memory>
#include <new> // Определение функции set_new_handler

// Обрабатываем ошибку выделения памяти
void customNewHandler() {
   std::cerr << "customNewHandler was called\n";
   std::exit(EXIT_FAILURE);
}

int main() {
   std::array<std::unique_ptr<double[]>, 1000> items{};

   // Указываем, что при ошибке выделения памяти
   // должна быть вызвана функция customNewHandler
   std::set_new_handler(customNewHandler);

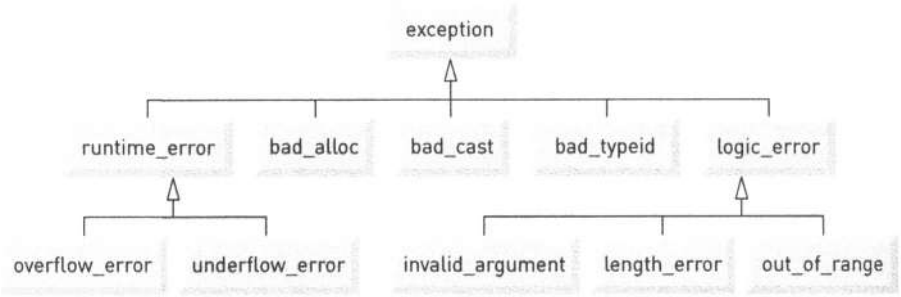
   // Направляем каждый unique_ptr на большой блок памяти
   for (int i{0}; auto& item : items) {
      item = std::make_unique<double[]>(500'000'000);
      std::cout << fmt::format(
         "items[{}] points to 500 000 000 doubles\n", i++);
   }
}
```

```
items[0] points to 500 000 000 doubles
items[1] points to 500 000 000 doubles
items[2] points to 500 000 000 doubles
items[3] points to 500 000 000 doubles
items[4] points to 500 000 000 doubles
items[5] points to 500 000 000 doubles
items[6] points to 500 000 000 doubles
items[7] points to 500 000 000 doubles
items[8] points to 500 000 000 doubles
customNewHandler was called
```

Илл. 12.7 | Функция set_new_handler задает функцию-обработчик исключений оператора new

12.9. Иерархия исключений стандартной библиотеки

Исключения логично делятся на несколько категорий. Стандартная библиотека C++ включает иерархию классов исключений, часть из которых показана на следующей диаграмме:



Список типов исключений стандартной библиотеки C++, содержащий, в частности, новые типы исключений из C++20 `nonexistent_local_time`, `ambiguous_local_time` и `format_error`, вы найдете здесь:

https://en.cppreference.com/w/cpp/error/exception

Вы можете создавать программы, которые генерируют:

- стандартные исключения;
- исключения, производные от стандартных исключений;
- ваши собственные исключения, не производные от стандартных исключений;
- экземпляры неклассовых типов, таких как основные типы и указатели.

Иерархия классов исключений является хорошей отправной точкой для создания пользовательских типов исключений. На практике вместо генерации исключений, показанных на предыдущей диаграмме, в C++ Core Guidelines рекомендуется генерировать исключения производных классов, специфичных для вашего приложения. Такие пользовательские типы более информативны, чем исключения с общими именами, например `runtime_error`.

Базовый класс exception

На вершине иерархии исключений стандартной библиотеки находится базовый класс `exception` (заголовок `<exception>`). Этот класс содержит виртуальную функцию `what`, которую производные классы могут переопределять для вывода сообщений об ошибках. Если обработчик `catch` определяет ссылку на исключение базового класса, он может перехватывать объекты всех классов исключений, открыто производных от этого базового класса[1].

Классы, производные от exception

Прямыми производными классами от `exception` являются классы `runtime_error` и `logic_error` (оба определены в заголовке `<stdexcept>`), каждый из которых имеет

[1] C++ Core Guidelines, «E.15: Catch Exceptions from a Hierarchy By Reference». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-exception-ref.

несколько собственных производных классов. Также производными от класса `exception` являются исключения, генерируемые операторами C++:

- `bad_alloc` генерируется оператором `new` (раздел 12.8);
- `bad_cast` генерируется оператором `dynamic_cast`;
- `bad_typeid` генерируется оператором `typeid`.

Классы, производные от runtime_error

Класс `runtime_error`, который мы поверхностно рассмотрели в разделах 12.2 и 12.5, является базовым для нескольких других стандартных классов исключений, которые указывают на ошибки во время выполнения:

- Класс `overflow_error` указывает на арифметическую ошибку переполнения (то есть результат, который больше наибольшего положительного числа или меньше наименьшего отрицательного числа, сохраняемого в переменной данного типа).
- Класс `underflow_error` указывает на ошибку арифметической неточности, которая может возникнуть при работе с «ненормальными числами с плавающей точкой»[1].

Классы, производные от logic_error

Класс `logic_error` является базовым для нескольких стандартных классов исключений, которые указывают на ошибки в логике программы:

- Мы использовали класс `invalid_argument` в *set*-функциях (начиная с главы 9), чтобы указать на попытку записи недопустимого значения. Правильно написанный код, конечно, может предотвратить попадание недопустимых аргументов в функцию.
- Класс `length_error` указывает на попытку операции со слишком большим объектом (размер которого больше максимально допустимого).
- Класс `out_of_range` указывает, что значение (например, индекс массива) вышло за пределы допустимого диапазона.

Перехват исключений, связанных наследованием

Наследование в исключениях позволяет обработчику `catch` перехватывать общие ошибки:

- Один из подходов заключается в перехвате каждого типа исключений производного класса по отдельности. Этот подход может приводить к ошибкам: вы можете забыть явно проверить один или несколько производных типов.

[1] «Subnormal Number». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Subnormal_number.

- Более лаконичный подход заключается в перехвате ссылок на объекты исключения базового класса.

Это будет логической ошибкой, если вы разместите обработчик `catch` для перехвата исключений базового класса перед обработчиком, который перехватывает исключения производного класса, — компиляторы выдадут предупреждение об этом. Обработчик `catch` базового класса может обрабатывать исключения всех классов, открыто производных от базового класса, поэтому в данном случае обработчик `catch` производного класса никогда не будет выполняться[1].

Перехват всех исключений

Исключения в C++ не обязательно должны быть производными от класса `exception`, поэтому перехват исключений, связанных с классом `exception`, не гарантирует перехвата всех исключений, которые могут случиться в программе. Вы можете использовать обработчик `catch(...)` для перехвата всех типов исключений, генерируемых в блоке `try`. У этого подхода есть слабые стороны:

- Тип перехваченного исключения остается неизвестным.
- Без именованного параметра вы не сможете ссылаться на объект исключения внутри обработчика исключений.

Обработчик `catch(...)` в основном используется для восстановления программы, которое не зависит от типа исключения, например для освобождения общих ресурсов. Исключение может генерироваться повторно, чтобы оповестить внешние обработчики `catch`.

12.10. Альтернатива блоку finally в языке C++: RAII («получение ресурсов есть инициализация»)

В некоторых языках программирования, созданных после C++ (например, Java, C# и Python), инструкция `try` имеет необязательный блок `finally`, гарантированно выполняющийся независимо от того, как завершится блок `try` — успешно или с исключением. Блок `finally` помещается или после последнего обработчика `catch`, или сразу после блока `try` при отсутствии обработчиков исключений (что разрешено в этих языках, но не в C++). Это делает блоки `finally` хорошим механизмом для гарантированного освобождения ресурсов и предотвращения их утечек.

Как программист, знающий другие языки программирования, вы можете задаться вопросом, почему инструкция `try` в языке C++ не содержит блок `finally`. В C++ не требуется блок `finally`, потому что вы можете использовать идиому RAII («по-

[1] C++ Core Guidelines, «E.31: Properly Order Your catch-Clauses». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re_catch/.

лучение ресурсов есть инициализация»), умные указатели и деструкторы. Если вы создаете класс в соответствии с RAII, объекты вашего класса будут получать нужные им ресурсы при создании объекта и освобождать их при уничтожении объекта.

С годами аналогичные возможности были добавлены в Java, C# и Python:

- в Java есть инструкции «`try` с ресурсами»;
- в C# есть инструкции `using`;
- в Python есть инструкции `with`.

Когда управление программой передается этим инструкциям, каждая из них создает объекты, получающие ресурсы, которые можно использовать в теле этих инструкций. Когда эти инструкции завершают управление — успешно или из-за возникшего исключения, — они автоматически освобождают ресурсы.

12.11. Библиотеки, поддерживающие и исключения, и коды ошибок

Исключения используются не повсеместно. Опрос разработчиков C++, проведенный в 2018 году, показал, что исключения частично или полностью запрещались в 52% проектов[1]. Например:

- в Google C++ Style Guide[2] сообщается, что корпорация Google не использует исключения;
- в стандартах программирования на C++ для Joint Strike Fighter[3] запрещено использовать инструкции `try`, `catch` и `throw`.

Когда организация запрещает работу с исключениями, она также запрещает использовать библиотеки с функциями, которые могут вызывать исключения, в том числе стандартную библиотеку C++.

У запрета исключений в проектах есть недостатки. Перечислим некоторые проблемы, с которыми может столкнуться разработчик в таком проекте[4]:

---

[1] «C++ Developer Survey "Lite": 2018–02», февраль 2018 г. Был доступен: 15 сентября 2023 г. https://isocpp.org/files/papers/CppDevSurvey-2018-02-summary.pdf.

[2] «Google C++ Style Guide». Был доступен: 15 сентября 2023 г. https://google.github.io/styleguide/cppguide.html.

[3] «Joint Strike Fighter Air Vehicle C++ Coding Standards», декабрь 2005 г. Был доступен: 15 сентября 2023 г. https://www.stroustrup.com/JSF-AV-rules.pdf.

[4] Ответ Лучана Раду Теодореску (Lucian Radu Teodorescu) на вопрос «Why do some people recommend not using exception handling in C++? Is this just a "culture" in C++ community, or do some real reasons exist behind this?», 2 августа 2015 г. Был доступен: 15 сентября 2023 г. https://www.quora.com/Why-do-some-people-recommend-not-using-exception-handling-in-C++-Is-this-just-a-culture-in-C++-community-or-do-some-real-reasons-exist-behind-this/answer/Lucian-Radu-Teodorescu.

- проблемы совместимости с библиотеками классов, использующими исключения;
- большой объем кода, необходимого для обработки ошибок;
- проблемы с ошибками при создании объектов;
- проблемы с перегруженными операторами присваивания;
- трудности с обработкой ошибок в потоках управления;
- загромождение кода из-за того, что его логика смешивается с обработкой ошибок;
- проблемы с распределением и освобождением ресурсов;
- снижение эффективности кода.

Чтобы предоставить программистам гибкие возможности решать, когда имеет смысл использовать исключения, некоторые библиотеки поддерживают двойные интерфейсы, предоставляя две версии каждой функции:

- версию, которая генерирует исключение при возникновении проблемы;
- версию, которая устанавливает или возвращает индикатор ошибки при возникновении проблемы.

В качестве примера можно рассмотреть функции из библиотеки `<filesystem>` (C++17), позволяющие управлять файлами и папками из приложений C++[1]. В этой библиотеке каждая функция имеет две версии: одну, генерирующую исключение `filesystem_error`, и другую, изменяющую значение переданного по ссылке аргумента `error_code`.

12.12. Журнал ошибок

Одной из распространенных задач при обработке исключений является запись точек их возникновения в журнал ошибок — понятный человеку текстовый файл, который разработчики могут анализировать для отладки приложения. Журнал ошибок полезен во время разработки для обнаружения и устранения ошибок. Он пригодится и после релиза приложения: в случае сбоя приложение будет вносить запись в журнал, который пользователь может отправить разработчику.

Работа с журналом ошибок поддерживается стандартной библиотекой C++, хотя вы можете создать собственные механизмы ведения журнала, используя имеющиеся в C++ средства работы с файлами. Кроме этого, есть библиотеки с открытым исходным кодом, предназначенные для журналирования ошибок в C++:

- Boost.Log — https://www.boost.org/doc/libs/1_75_0/libs/log/doc/html;
- Easylogging++ — https://github.com/amrayn/easyloggingpp;

[1] «Filesystem Library». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/filesystem.

- Google Logging Library (glog) — https://github.com/google/glog;
- Loguru — https://github.com/emilk/loguru;
- Plog — https://github.com/SergiusTheBest/plog;
- spdlog — https://github.com/gabime/spdlog.

Некоторые из них являются заголовочными библиотеками, которые вы можете просто включить в свои проекты, другие требуют процедур установки.

12.13. Основы контрактного программирования[1]

Для улучшения архитектуры обработки ошибок вы можете указать ожидаемые состояния до и после выполнения каждой функции с помощью предусловий или постусловий соответственно. Мы определим их, покажем примеры кода с предусловиями и объясним, что происходит при нарушении контрактов.

- Предусловие[2] должно быть истинным при вызове функции. Предусловия описывают ограничения параметров функций или любые другие параметры, которые функция получает непосредственно перед началом выполнения. Если предусловие не выполняется, то поведение функции будет неопределенным: она может выдать исключение, продолжить работу с недопустимыми значениями или попытаться восстановиться после ошибки. Если разрешить выполнение кода с неопределенным поведением, программа будет вести себя непредсказуемо, особенно при портировании на другие платформы. Каждая функция может иметь несколько предусловий.
- Постусловие[3] должно быть истинным после успешного завершения функции. Постусловия описывают гарантии относительно возвращаемого значения или других параметров, на которые влияет выполнение функции. При определении функции вы должны документировать все постусловия, чтобы другие программисты понимали, чего следует ожидать при вызове вашей функции. Вы также должны гарантировать, что ваша функция выполняет все постусловия, если выполнены все предусловия. Каждая функция может иметь несколько постусловий.

Нарушения предусловий и постусловий

В последние годы нарушения предусловий и постусловий часто устраняются методом генерации исключений. Рассмотрим функцию `at` классов `array` и `vector`, которая получает индекс контейнера. В качестве предусловия функция `at` требует, чтобы аргумент `index` был больше или равен `0` и меньше размера контейнера. Если это предусловие выполняется, постусловие функции `at` указывает, что функция

---

[1] Контракты еще не добавлены в стандарт C++. Синтаксис, который мы показываем, может измениться.

[2] «Precondition». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Precondition.

[3] «Postcondition». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Postcondition.

вернет элемент с индексом `index`, в противном случае функция `at` выдает исключение `out_of_range`. Как клиент класса `array` или `vector`, вы верите в истинность постусловия функции `at`, если вы выполнили предусловие.

Предусловия и постусловия являются утверждениями, поэтому их можно реализовать с помощью макроса препроцессора `assert` (раздел 12.6.1), когда управление программой передается функции или возвращается из нее. Макросы препроцессора уже редко используются в современном C++. Но до тех пор, пока контракты не станут полноценной частью C++, вы будете продолжать использовать макрос `assert` или свой собственный код для реализации предусловий и постусловий.

Инварианты

Инвариант — это условие, которое всегда должно быть истинным в вашем коде, то есть условие, которое никогда не меняется. Инварианты класса должны быть истинными для каждого объекта класса. Как правило, они привязаны к жизненному циклу объекта. Инварианты класса остаются верными с момента создания объекта до его уничтожения. Например:

- класс `Account` из раздела 9.6 требует, чтобы его элемент данных `m_balance` всегда был неотрицательным;
- класс `Time` из раздела 9.7 требует, чтобы его элемент данных `m_hour` всегда имел значение в диапазоне от `0` до `23`, а его элементы `m_minute` и `m_second` всегда имели значения в диапазоне от `0` до `59`.

Эти инварианты гарантируют, что объекты классов `Account` и `Time` содержат допустимые данные на протяжении всего жизненного цикла.

Функции также могут содержать инварианты. Например, если функция выполняет поиск значения в векторе `vector<int>`, инвариантом будет следующее условие: если в векторе есть искомое значение, его индекс должен быть не меньше `0` и меньше размера вектора.

Контрактное проектирование

Контрактное проектирование (design by contract, или сокращенно **DbC**)[1,2,3] — это подход к разработке программного обеспечения, созданный Бертраном Мейером в 1980-х годах и использованный при разработке его языка программирования Eiffel. Основные принципы этого подхода:

- функция предполагает, что клиентский код будет соответствовать предусловиям;

---

[1] Бертран Мейер (Bertrand Meyer). «Object-Oriented Software Construction». Изд. Prentice Hall, 1988.

[2] Бертран Мейер, «Touch of Class: Learning to Program Well with Objects and Contracts», xvii. Издательство Springer Berlin AN, 2016.

[3] «Design by Contract». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Design_by_contract.

- если предусловия выполняются, функция гарантирует, что ее постусловия тоже будут выполнены;
- поддерживаются любые инварианты.

Добавить в стандарт C++ поддержку контрактного программирования (часто называемого «контрактами») впервые предложили в 2012 году, но это предложение было отклонено[1]. Затем было утверждено предложение включить контракты в C++20, но впоследствии, в конце разработки C++20, его отменили[2] из-за «сохраняющихся разногласий и проблем с проектированием»[3]. Таким образом, контракты отложены по крайней мере до C++23.

Постепенный переход к контрактам в стандартной библиотеке C++

Херб Саттер говорит: «В Java и .NET около 90% всех исключений генерируется из-за нарушения предусловий». Он также говорит: «Мир программирования уже признает, что ошибки программирования (например, попытки доступа за границы контейнеров, разыменование указателей `null`, нарушения всех предусловий, постусловий и инвариантов) приводят к сбоям, которые невозможно восстановить в режиме выполнения, поэтому не надо передавать в вызывающий код соответствующие исключения или коды ошибок, которые могут быть каким-то образом обработаны»[4].

Многие ошибки, которые в настоящее время обрабатываются в виде исключений, могут быть обнаружены с помощью предусловий и постусловий и устранены путем исправления кода — это и есть ключевая идея внедрения контрактов[5].

Цель контрактов — сделать большинство функций не выдающими исключения (`noexcept`)[6], что позволит компилятору выполнять дополнительные оптимизации. Саттер говорит: «Постепенный перенос нарушения предусловий с исключений на контракты обещает в конечном итоге устранить большую часть всех исключений, создаваемых стандартной библиотекой»[7,8].

---

[1] Натан Майерс (Nathan Meyers), «What Happened to C++20 Contracts?», 5 августа 2019 г. Был доступен: 15 сентября 2023 г.

[2] Натан Майерс, «What Happened to C++20 Contracts?».

[3] Херб Саттер, «Trip Report: Summer ISO C++ Standards Meeting (Cologne)», июль 2019 г. Был доступен: 15 сентября 2023 г. https://herbsutter.com/2019/07/20/trip-report-summer-iso-c-standards-meeting-cologne/.

[4] Херб Саттер, «Trip Report: Summer Iso C++ Standards Meeting (Rapperswil)», июль 2018 г. Был доступен: 15 сентября 2023 г. https://herbsutter.com/2018/07/.

[5] Гленнан Карни (Glennen Carnie), «Contract Killing (in Modern C++)», 18 сентября 2019 г. Был доступен: 15 сентября 2023 г. https://blog.feabhas.com/2019/09/contract-killing-in-modern-c/.

[6] Херб Саттер (Herb Sutter), «Trip report: Summer ISO C++ standards meeting» (Rapperswil).

[7] Херб Саттер, «Trip report: Summer ISO C++ standards meeting» (Rapperswil).

[8] Чтобы получить представление о масштабе исключений, генерируемых C++ и его библиотеками, мы провели поиск слова «throws» в окончательном проекте стандарта C++, расположенном по адресу https://isocpp.org/files/papers/N4860.pdf. Объем документа — более 1800 страниц; свыше 450 страниц посвящено языку, более 1000 — стандартной библиотеке,

Атрибуты контрактов

Предлагаемая[1] сигнатура объявления контракта такова:

> [[*атрибутКонтракта опциональныйУровень опциональныйИдентификатор: условие*]]

Вы можете использовать ее для указания предусловий, постусловий и утверждений для ваших функций. Идентификатор — это одна из локальных переменных функции для использования в постусловиях, как будет показано на илл. 12.9. Атрибутами контракта могут быть:

- `expects` — указывает на предусловие функции, которое проверяется перед началом выполнения тела функции;
- `ensures` — указывает на постусловие функции, которое проверяется непосредственно перед возвратом из функции;
- `assert` — указывает на утверждение, которое проверяется по мере появления во время выполнения функции.

Если вы задаете несколько предусловий и постусловий, они проверяются в порядке их объявления.

Уровни контрактов

Есть три уровня контрактов:

- `default` — определяет контракт с небольшой нагрузкой на компьютер во время выполнения (функция выполняется чуть медленнее, чем обычно). Если уровень контракта не указан, компилятор автоматически применяет `default`;
- `audit` — определяет контракт со значительной нагрузкой на компьютер во время выполнения (функция выполняется заметно медленнее, чем обычно). Такие контракты предназначены в первую очередь для применения в процессе разработки программы;
- `axiom` — определяет контракт, предназначенный для принудительного исполнения с помощью статических средств проверки кода, а не во время выполнения.

а остальные — приложениям, библиографии, перекрестным ссылкам и указателям. Слово «throws» встречается 422 раза, в том числе 92 — «throws nothing», 13 — «throws nothing unless...» (указывает на исключение, выдаваемое при размотке стека) и 329 упоминаний относятся к функциям, генерирующим исключения. Многие из этих 329 упоминаний являются примерами того, как контракты помогут устранить необходимость в исключениях.

[1] Г. Дос Рейс (G. Dos Reis), Дж. Д. Гарсия (J. D. Garcia), Дж. Лакос (J. Lakos), А. Мередит (A. Meredith), Н. Майерс (N. Meyers) и Б. Страуструп (B. Stroustrup), «Support for Contract Based Programming in C++», 8 июня 2018 г. Был доступен: 15 сентября 2023 г. http://www.openstd.org/jtc1/sc22/wg21/docs/papers/2018/p0542r5.html.

Эти уровни позволяют выбирать, какие контракты будут применяться во время выполнения, чтобы контролировать нагрузку на компьютер. Вы можете выбрать — предположительно с помощью флагов компилятора, — следует ли полностью отключить контракты, или выполнять `default`-контракты с малой нагрузкой, или выполнять `audit`-контракты с большей нагрузкой.

Определение предусловий, постусловий и утверждений

Контракты с предусловием (`expects`) и постусловием (`ensures`) указываются в прототипе функции, они перечисляются после сигнатуры функции и перед точкой с запятой. Например, функция, которая вычисляет действительный (а не мнимый) квадратный корень из значения `double`, ожидает, что ее аргумент будет больше или равен `0`. Вы можете указать это с помощью контракта с предусловием:

```
double squareRoot(double value)
    [[expects: value >= 0.0]];
```

Если определение функции также служит прототипом функции, контракты с предусловием и постусловием перечисляются между сигнатурой функции и открывающей левой фигурной скобкой.

Утверждения указываются как инструкции в теле функции. Например, утверждение, проверяющее нахождение целочисленной оценки за экзамен (переменная `grade`) в диапазоне от `0` до `100`, объявляется следующим образом:

```
[[assert: grade >= 0 && grade <= 100]];
```

Обратите внимание, что утверждение в этой инструкции не является макросом `assert`.

Ранняя реализация контрактов

В компиляторе GNU C++ уже есть ранняя реализация контрактов[1]. Вы можете протестировать ее на веб-сайте Compiler Explorer (https://godbolt.org)[2], который поддерживает множество версий компиляторов и языков программирования. Вы можете выбрать `x86-64 gcc (contracts)` в качестве своего компилятора и скомпилировать код с контрактами, используя параметры компилятора, описанные по ссылке:

https://gitlab.com/lock3/gcc-new/-/wikis/contract-assertions

Мы предоставляем адрес godbolt.org, где вы можете протестировать каждый из наших примеров, используя раннюю реализацию. В ранней реализации контрактов в GNU C++ используются другие ключевые слова для предусловий и постусловий:

- `pre` вместо `expects`;
- `post` вместо `ensures`.

---

[1] Есть также реализация контрактов для компилятора Clang (https://github.com/arcosuc3m/clang-contracts), но вам нужно собрать и установить эту программу самостоятельно.

[2] Copyright © 2012–2019, авторы Compiler Explorer. Все права защищены. Compiler Explorer принадлежит Мэтту Годболту (Matt Godbolt). https://xania.org/MattGodbolt.

Пример с делением на ноль

На илл. 12.8 показана переопределенная функция `quotient`, представленная на илл. 12.3. В прототипе функции (строки 5–6) мы объявляем контракт с уровнем `default`, где переменная `denominator` не должна быть равна `0.0`. Ключевое слово `default` также можно указывать явно:

```
[[pre default: denominator != 0.0]]
```

Этот пример расположен по адресу:

https://godbolt.org/z/jn4MK3o9T

Когда вы вводите код с клавиатуры или загружаете его в Compiler Explorer, компилятор автоматически компилирует и запускает его или отображает ошибки компиляции.

```
 1 // fig12_08.cpp
 2 // Функция quotient с предусловием контракта
 3 #include <iostream>
 4
 5 double quotient(double numerator, double denominator)
 6    [[pre: denominator != 0.0]];
 7
 8 int main() {
 9    std::cout << "quotient(100, 7): " << quotient(100, 7)
10       << "\nquotient(100, 0): " << quotient(100, 0) << '\n';
11 }
12
13 // Выполняем деление
14 double quotient(double numerator, double denominator) {
15    return numerator / denominator;
16 }
```

Илл. 12.8 | Функция quotient с предусловием контракта

Для этого примера мы использовали следующий параметр компилятора:

```
-std=c++20 -fcontracts
```

Этот параметр компилирует код с поддержкой C++20 и экспериментальных контрактов. Compiler Explorer обычно показывает по крайней мере две вкладки: редактор кода и компилятор. В наших примерах также используется вкладка вывода, на которой показаны результаты выполняемой программы и сообщения об ошибках компиляции. Вкладка компилятора имеет два выпадающих списка вверху. Один из них позволяет вам выбрать компилятор, другой — просматривать и редактировать параметры компилятора. Вы также можете нажать на «стрелку вниз», чтобы открыть общие параметры компилятора.

Вызов функции `quotient` в строке 9 соответствует предусловию и успешно выполняется, выводя результат:

```
quotient(100, 7): 14.2857
```

Однако вызов функции `quotient` в строке 10 приводит к нарушению контракта. Обработчик нарушений по умолчанию (`handle_contract_violation`) вызывается неявно, выводя сообщение об ошибке и завершая работу программы:

```
default std::handle_contract_violation called:
./example.cpp 6 quotient denominator != 0.0 default default 0
```

Параметр компилятора:

```
-std=c++20 -fcontracts -fcontract-build-level=off
```

отключает проверку контракта и позволяет выполнить строку 10 с таким результатом:

```
quotient(100, 0): inf
```

Деление на ноль в C++ приводит к неопределенному поведению, но многие компиляторы, включая три компилятора, которые мы используем в этой книге, возвращают положительную бесконечность (`inf`) или отрицательную бесконечность (`-inf`). Это поведение определено стандартом IEEE 754 для арифметики с плавающей точкой, который широко поддерживается современными языками программирования.

Режим продолжения

По умолчанию **режим продолжения (continuation mode)** при нарушении контракта немедленно завершает работу программы. Чтобы разрешить программе продолжать выполнение, вы можете добавить такую опцию компилятора:

```
-fcontract-continuation-mode=on
```

Пример: функция поиска, требующая отсортированный вектор

Рассмотрим илл. 12.9, где определен шаблон функции `binarySearch` с двумя предусловиями: принимаемый в качестве аргумента вектор должен содержать элементы и должен быть отсортирован. Вы можете протестировать этот пример по адресу:

https://godbolt.org/z/obMsY5Wo4

В этом примере мы определяем функцию `binarySearch` перед функцией `main` и помещаем все предусловия после списка параметров и перед телом функции.

```
 1 // fig12_09.cpp
 2 // Функция binarySearch с предусловием, требующим отсортированный вектор
 3 #include <algorithm>
 4 #include <iostream>
 5 #include <vector>
 6
 7 template<typename T>
 8 int binarySearch(const std::vector<T>& items, const T& key)
 9    [[pre: items.size() > 0]]
10    [[pre audit: std::is_sorted(items.begin(), items.end())]] {
```

Окончание ⇨

```
11    size_t low{0}; // Нижний индекс элементов для поиска
12    size_t high{items.size() - 1}; // Верхний индекс
13    size_t middle{(low + high + 1) / 2}; // Средний элемент
14    int loc{-1}; // Индекс ключа; -1, если элемент не найден
15
16    do { // Цикл для поиска элемента
17       // Если элемент найден в середине
18       if (key == items[middle]) {
19          loc = middle; // loc - середина в данный момент
20       }
21       else if (key < items[middle]) { // Середина находится слишком высоко
22          high = middle - 1; // Устраняем верхнюю половину
23       }
24       else { // Середина находится слишком низко
25          low = middle + 1; // Устраняем нижнюю половину
26       }
27
28          middle = (low + high + 1) / 2; // Пересчитываем середину
29       } while ((low <= high) && (loc == -1));
30
31    return loc; // Возвращаем расположение key
32 }
33
34 int main() {
35    // Отсортированный вектор v1 выполняет предусловие функции binarySearch
36    std::vector v1{10, 20, 30, 40, 50, 60, 70, 80, 90};
37    int result1{binarySearch(v1, 70)};
38    std::cout << "70 was " << (result1 != -1 ? "" : "not ")
39       << "found in v1\n";
40
41    // Неотсортированный вектор v2 нарушает предусловие функции binarySearch
42    std::vector v2{60, 70, 80, 90, 10, 20, 30, 40, 50};
43    int result2{binarySearch(v2, 60)};
44    std::cout << "60 was " << (result2 != -1 ? "" : "not ")
45       << "found in v2\n";
46 }
```

Илл. 12.9 | Функция binarySearch с предусловием и постусловием

Шаблон функции `binarySearch` выполняет бинарный поиск по вектору, который должен быть отсортирован в порядке возрастания; в противном случае результат может быть ошибочным. Реализация функции также требует, чтобы вектор содержал элементы. Итак, мы объявили два предусловия:

```
[[pre: items.size() > 0]]
[[pre audit: is_sorted(begin(items), end(items))]]
```

Первое предусловие гарантирует, что вектор содержит элементы. Второе вызывает стандартную функцию библиотеки C++ `is_sorted` (заголовок `<algorithm>`)[1] для про-

[1] В заголовке <algorithm> стандартной библиотеки C++ есть более 200 функций. Мы рассмотрим многие из них в главе 14.

верки сортировки вектора. Эта операция снижает быстродействие. Двоичный поиск отсортированного вектора из миллиарда элементов требует всего 30 сравнений. Но чтобы выяснить, отсортирован ли этот вектор, требуется `999 999 999` сравнений. А эффективная сортировка одного миллиарда элементов с помощью алгоритма сортировки $O(n \log n)$ потребовала бы около 30 миллиардов операций. Разработчик может включить этот тест во время разработки и отключить его в финальном коде. По этой причине мы указали предусловие как `audit` на уровне контракта.

Для этого примера мы использовали такие параметры компилятора:

```
-std=c++2a -fcontracts
```

Они означают поддержку C++20 и контрактов с уровнем `default`, поэтому контракт предусловия с уровнем `audit` в строке 10 игнорируется. В функции `main` мы создали два целочисленных вектора, содержащих одинаковые значения: отсортированный `v1` (строка 36) и неотсортированный `v2` (строка 42). При начальных настройках компилятора предусловие, обеспечивающее отсортированный вектор, не проверяется, поэтому вызовы бинарного поиска в строках 16 и 21 завершаются и программа отображает выходные данные:

```
70 was found in v1
60 was not found in v2
```

Поскольку предусловие с уровнем `audit` было проигнорировано, в нашей программе возникла логическая ошибка. Вторая строка вывода показывает, что число `60` не найдено в векторе `v2`. Алгоритм ожидает, что `v2` будет отсортирован, поэтому ему не удалось найти число `60`, хотя оно есть в векторе. Изменение уровня сборки контракта на `audit` в параметрах компиляции

```
-std=c++2a -fcontracts -fcontract-build-level=audit
```

включает проверку контрактов с уровнем `audit`. Когда программа запускается с этим параметром, вектор `v1` в вызове функции `binarySearch` в строке 37 соответствует предусловию из строки 10, поэтому в строках 38–39 снова выводится сообщение:

```
70 was found in v1
```

Однако вектор `v2` в вызове функции `binarySearch` в строке 43 нарушает контракт, что приводит к сообщению об ошибке:

```
default std::handle_contract_violation called:
./example.cpp 10 binarySearch<int> is_sorted(begin(items), end(items)) audit
default 0
```

Обработчик нарушений пользовательского контракта

В приведенных примерах использовалась стандартная обработка нарушений контракта. Если контракт нарушен, создается объект `contract_violation`, содержащий информацию о том, где произошло нарушение:

- номер строки, возвращаемый функцией `line_number`;
- имя файла исходного кода, возвращаемое функцией `file_name`;
- имя функции, возвращаемое функцией `function_name`;
- описание нарушенного условия, возвращаемое функцией `comment`;
- уровень контракта, возвращаемый функцией `assertion_level`.

Объект `contract_violation` передается в функцию обработки нарушений:

```
void handle_contract_violation(const contract_violation& violation) {
   // Код обработчика
}
```

Экспериментальная реализация контрактов в g++ предоставляет реализацию этой функции по умолчанию. Если вы определите пользовательскую версию, компилятор g++ ее вызовет.

12.14. Итоги

Язык C++ используется для создания критически важного коммерческого программного обеспечения. Многие системы, написанные на C++, крайне сложны. В этой главе вы узнали, что очень важно устранять ошибки во время разработки и предусматривать способы решения проблем, выявляемых после внедрения программного обеспечения.

Мы рассмотрели случаи появления исключений в вашем коде. Мы объяснили, как обработка исключений помогает писать надежные, отказоустойчивые программы, которые обрабатывают редкие ошибки и продолжают работать, обеспечивая корректное завершение с освобождением ресурсов, или немедленно завершают работу в случае непредвиденных исключений.

Мы рассмотрели концепции обработки исключений в примере, который демонстрировал потоки управления при успешном выполнении программы и при возникновении исключения. Мы обсудили варианты перехвата и повторного создания исключений. Вы узнали, как размотка стека позволяет другим функциям обрабатывать исключения, не перехваченные в определенной области видимости.

Вы узнали, когда исключения необходимы. Мы также рассмотрели варианты гарантии безопасности исключений, которые вы можете предоставлять в своем коде: отсутствие гарантии, базовая гарантия, строгая гарантия и гарантия отсутствия исключений. Вы узнали, почему исключения нужны для выявления ошибок в конструкторах и почему не должно быть исключений в деструкторах. Мы показали, как использовать функциональные блоки `try` для перехвата исключений из списка инициализаторов конструктора или из деструкторов базового класса при уничтожении объекта производного класса.

Вы узнали, что оператор `new` выдает исключение `bad_alloc` при сбое выделения памяти, а в устаревшем коде C++ ошибки выделения памяти обрабатываются с помощью функции `set_new_handler`.

Вы познакомились с иерархией классов исключений стандартной библиотеки C++ и создали пользовательский класс исключений, производный от класса исключений стандартной библиотеки C++. Вы узнали, почему важно перехватывать исключения по ссылке, что позволяет обработчикам исключений перехватывать связанные по наследованию типы исключений без среза. Вы также узнали о том, как вести журнал ошибок, чтобы затем анализировать его, отлаживая приложение.

Вы узнали, почему в некоторых организациях запрещены исключения. Вы также узнали, что некоторые библиотеки предоставляют двойные интерфейсы, чтобы разработчики могли выбирать между версиями функций, генерирующими исключения, и версиями, использующими коды ошибок.

Мы завершили главу кратким объяснением принципов контрактного программирования, поддержка которого, первоначально запланированная в C++20, перенесена в следующие версии C++. Мы показали, как использовать контракты для тестирования предусловий, постусловий и утверждений во время выполнения программы. Вы узнали, что эти условия всегда должны быть истинными в правильном коде. Если условия являются ложными, контракт нарушается и по умолчанию программа немедленно завершается. Это позволяет быстрее находить ошибки, устранять их во время разработки и создавать более надежный код.

В главе 6 мы рассказали о классах `array` и `vector` стандартной библиотеки C++. В главе 13 вы узнаете о других контейнерах стандартной библиотеки, а также об итераторах, которые нужны алгоритмам стандартной библиотеки для того, чтобы перебирать элементы контейнеров и оперировать их элементами.

13

Контейнеры и итераторы стандартной библиотеки

В этой главе вы:

- больше узнаете о контейнерах, итераторах и алгоритмах стандартной библиотеки C++;
- узнаете о взаимосвязи контейнеров с диапазонами C++20;
- научитесь использовать итераторы потоков ввода-вывода для чтения из стандартного потока ввода и для записи в стандартный поток вывода;
- научитесь использовать итераторы для доступа к элементам контейнеров;
- научитесь работать с последовательными контейнерами `vector`, `list` и `deque`;
- научитесь использовать `ostream_iterator` с алгоритмами `std::copy` и `std::ranges::copy` для вывода элементов контейнера одной инструкцией;
- познакомитесь с упорядоченными ассоциативными контейнерами `set`, `multiset`, `map` и `multimap`;
- узнаете о различиях между упорядоченными и неупорядоченными ассоциативными контейнерами;
- научитесь использовать адаптеры контейнеров `stack`, `queue` и `priority_queue`;
- примените псевдоконтейнеры `bitset` для обработки набора битовых флагов.

13.1. Введение

В стандартной библиотеке есть основанные на шаблонах мощные компоненты, реализующие множество структур данных и алгоритмов для их обработки. Мы начали рассказывать о шаблонах в главах 5–6 и будем часто применять их в главах 13–15. Исторически сложилось, что языковые средства, представленные в этой главе, относятся к стандартной библиотеке шаблонов (Standard Template Library — STL)[1]. В стандарте C++ они считаются частью стандартной библиотеки C++.

Контейнеры, итераторы и алгоритмы

В этой главе рассматриваются три ключевых компонента стандартной библиотеки: контейнеры (шаблонные структуры данных), итераторы и алгоритмы. Мы представим вам контейнеры, адаптеры контейнеров и псевдоконтейнеры.

Общие функции контейнеров

У каждого контейнера есть соответствующие функции класса. Некоторые из них определены во всех классах контейнеров. Мы рассматриваем большую часть этой общей функциональности в примерах с контейнерами `array` (представленными в главе 6), `vector` (представленными в главе 6 и подробно обсуждаемыми в этой главе), `list` (раздел 13.7) и `deque` (произносится «дэк»; раздел 13.8).

Итераторы

Итераторы (iterators), имеющие аналогичные с указателями свойства, используются для обработки элементов контейнеров. Массивы в стиле C тоже можно обрабатывать алгоритмами стандартной библиотеки, используя указатели в качестве итераторов. Вы увидите, как велики возможности обработки контейнеров с помощью итераторов в сочетании с и алгоритмами стандартной библиотеки: иногда множество строк кода удается сократить до одной инструкции.

Алгоритмы

Алгоритмы (algorithms) стандартной библиотеки (мы детально займемся ими в главе 14) — это шаблоны функций, выполняющих базовые операции с данными: поиск, сортировку, копирование, преобразование и сравнение элементов или целых контейнеров. Стандартная библиотека предоставляет много алгоритмов:

- 90 алгоритмов в заголовке `<algorithm>`, в пространстве имен `std`; 82 из них также перегружены в пространстве имен `std::ranges` для совместимости с диапазонами C++20;

---

[1] STL была разработана Александром Степановым (Alexander Stepanov) и Мэн Ли (Meng Lee) из Hewlett-Packard и основана на их исследованиях в области обобщенного программирования при значительном вкладе Дэвида Массера (David Musser). Степанов впервые предложил включить STL в C++ на заседании Комитета по стандартизации ANSI/ISO C++ в ноябре 1993 г. Заявка была одобрена в июле 1994 г. https://en.wikipedia.org/wiki/History_of_the_Standard_Template_Library. Был доступен: 15 сентября 2023 г.

- 11 алгоритмов в заголовке `<numeric>`, в пространстве имен `std`;
- 14 алгоритмов в заголовке `<memory>`, в пространстве имен `std`; все они также перегружены в пространстве имен `std::ranges` для совместимости с диапазонами C++20;
- 2 алгоритма в заголовке `<cstdlib>`.

Многие из этих алгоритмов введены в C++11 и C++20, а некоторые — в C++17. Здесь вы найдете полный список алгоритмов со ссылками на их описания:

https://en.cppreference.com/w/cpp/algorithm

Большинство алгоритмов использует итераторы для доступа к элементам контейнера. Каждый алгоритм предъявляет минимальные требования к тем типам итераторов, которые могут с ним работать. Мы покажем, что контейнеры поддерживают несколько типов итераторов, часть из которых мощнее, чем другие. Итераторы, поддерживаемые контейнером, определяют возможность использования этого контейнера с конкретным алгоритмом. Итераторы инкапсулируют механизмы работы с контейнерами и доступа к их элементам, позволяя применять многие алгоритмы со значительно отличающимися реализациями. Это также позволяет вам создавать новые алгоритмы, совместимые с контейнерами разных типов.

Диапазоны C++20

В главе 6 вы познакомились с **диапазонами (ranges)** и **представлениями (views)**, введенными в C++20. Вы узнали, что диапазон — это коллекция элементов, которые можно перебирать итерациями, поэтому массивы (`array`) и векторы (`vector`) — это диапазоны. Вы также использовали представления для создания конвейеров операций, управляющих диапазоном элементов. Любой контейнер с итераторами, представляющими его начало и конец, можно рассматривать как диапазон C++20. В этой главе мы будем использовать новый алгоритм стандартной библиотеки C++20 `std::ranges::copy` и более старый алгоритм стандартной библиотеки C++ `std::copy`, чтобы вы увидели, как диапазоны упрощают создание кода. В главе 14 мы расскажем о многих алгоритмах C++20 из пространства имен `std::ranges`, демонстрируя разнообразные возможности диапазонов и представлений.

Пользовательские структуры данных на основе шаблонов

Стандартная библиотека позволяет создавать такие полезные структуры данных, как связные списки, очереди, стеки и бинарные деревья.

- **Связный список (linked list)** — это набор элементов данных, логически «выстроенных в ряд». Вставки и удаления элементов возможны в любом месте списка.
- **Стек (stack)** — это структура данных, особенно необходимая в компиляторах и операционных системах. Вставки и удаления элсментов производятся только на одном конце стека — сверху.

- **Очередь (queue)** — это линейная последовательность элементов, которые чего-либо ожидают. Вставки элементов производятся с конца (с хвоста), а извлечения — спереди (с головы).
- **Бинарное дерево (binary tree)** — это нелинейная иерархическая структура данных, облегчающая поиск и сортировку данных, а также удаление дубликатов.

Все эти структуры данных имеют множество интересных применений. Вы могли бы структурировать данные и по-другому, тщательно формируя связи между объектами с помощью указателей, но код, основанный на указателях, сложен и подвержен ошибкам. Малейшие упущения или оплошности могут вызвать ошибки доступа к памяти и утечки памяти без предупреждения от компилятора.

Не надо «изобретать велосипед». При каждой возможности используйте готовые контейнеры, итераторы и алгоритмы стандартной библиотеки C++[1]. Готовые классы контейнеров предоставляют вам структуры данных, которых вполне достаточно для большинства приложений. Надежные контейнеры, итераторы и алгоритмы стандартной библиотеки сокращают время тестирования и отладки. Все эти компоненты языка C++ были задуманы и спроектированы с учетом требований быстродействия и гибкости.

13.2. Основы контейнеров[2]

Контейнеры стандартной библиотеки делятся на четыре основные категории:

- последовательные контейнеры;
- упорядоченные ассоциативные контейнеры;
- неупорядоченные ассоциативные контейнеры;
- адаптеры контейнеров.

Сначала мы кратко опишем контейнеры, а в следующих разделах покажем много практических примеров кода с ними.

Последовательные контейнеры

Последовательные контейнеры — это линейные структуры данных, все элементы которых концептуально «выстроены в ряд», например массивы, векторы и связные списки, которые поддаются сортировке. Есть пять стандартных последовательных контейнеров:

- `array` (глава 6) — контейнер фиксированного размера. Элементы расположены в памяти последовательно. Прямой доступ (произвольный доступ) к любому элементу.

---

[1] C++ Core Guidelines, «SL.1: Use Libraries Wherever Possible». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rsl-lib.

[2] Этот раздел задуман как введение к последующим разделам. Возможно, вы захотите быстро просмотреть его и впоследствии вернуться к нему при изучении примеров реального кода из этой главы.

- `vector` (раздел 13.6) — контейнер изменяемого размера. Элементы расположены в памяти последовательно. Быстрая вставка и удаление элементов с конца контейнера. Прямой доступ к любому элементу.
- `list` (раздел 13.7) — двусвязный список изменяемого размера. Быстрая вставка и удаление элементов в любом месте.
- `deque` (раздел 13.8) — контейнер изменяемого размера. Быстрая вставка и удаление элементов с начала или с конца. Прямой доступ к любому элементу.
- `forward_list` — односвязный список изменяемого размера. Быстрая вставка и удаление элементов в любом месте.

Связные списки не поддерживают произвольный доступ к любому элементу. Класс `string` похож по функциональности на последовательные контейнеры, но предназначен только для символьных данных.

Ассоциативные контейнеры

Ассоциативные контейнеры — это нелинейные структуры данных (обычно реализуемые в виде бинарных деревьев[1, 2]), обеспечивающие быстрый доступ к элементам. Такие контейнеры могут хранить наборы значений или пары «ключ — значение», где каждому ключу соответствует значение. Например, программа может связать числовые идентификаторы (ID) сотрудников с объектами класса `Employee`. Как вы увидите, некоторые ассоциативные контейнеры допускают связь каждого ключа с несколькими значениями. Ключи в ассоциативных контейнерах являются неизменяемыми: их невозможно модифицировать, не удалив предварительно из контейнера. Есть четыре стандартных упорядоченных ассоциативных контейнера:

- `multiset` (раздел 13.9.1) — поддерживает быстрый поиск, возможны дубликаты данных.
- `set` (раздел 13.9.2) — поддерживает быстрый поиск, дубликаты данных не допускаются.
- `multimap` (раздел 13.9.3) — поддерживает быстрый поиск по ключу, возможны дубликаты ключей. Сопоставление «один ко многим».
- `map` (раздел 13.9.4) — поддерживает быстрый поиск по ключу, дубликаты ключей не допускаются. Сопоставление «один к одному».

Есть четыре неупорядоченных ассоциативных контейнера (реализованных через хеширование[3, 4]):

---

[1] «set». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/container/set.

[2] «map». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/container/map.

[3] «unordered_set». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/container/unordered_set.

[4] «unordered_map»/ Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/container/unordered_map.

- `unordered_multiset` — поддерживает быстрый поиск, возможны дубликаты данных.
- `unordered_set` — поддерживает быстрый поиск, дубликаты данных не допускаются.
- `unordered_multimap` — сопоставление «один ко многим», возможны дубликаты данных, быстрый поиск по ключу.
- `unordered_map` — сопоставление «один к одному», дубликаты данных не допускаются, быстрый поиск по ключу.

Адаптеры контейнеров

Стандартная библиотека предоставляет вам `stack`, `queue` и `priority_queue` — адаптеры контейнеров, которые позволяют программе ограниченно просматривать последовательные контейнеры. Есть три адаптера контейнеров:

- `stack` — «последним пришел, первым ушел» (last-in, first-out — LIFO);
- `queue` — «первым пришел, первым ушел» (first-in, first-out — FIFO);
- `priority_queue` — приоритетная очередь: первым всегда уходит элемент с высшим приоритетом.

Псевдоконтейнеры

В C++ есть и другие типы контейнеров, которые правильнее называть **псевдоконтейнерами (near containers)**. К ним относятся массивы в стиле C (глава 7), класс `bitset` (раздел 13.11) для работы с отдельными битами, класс `string` и класс `valarray` для высокоскоростных математических векторных операций[1] (не путайте с контейнером `vector`). Эти типы нельзя считать полноценными контейнерами C++, потому что они обладают не всеми, а лишь некоторыми возможностями последовательных и ассоциативных контейнеров.

13.2.1. Общие вложенные типы в последовательных и ассоциативных контейнерах

В следующей таблице перечислены общие **вложенные типы (nested types)**, определенные во всех классах последовательных и ассоциативных контейнеров. Они используются в объявлениях шаблонных переменных, в параметрах функций и в возвращаемых функциями значениях, как будет показано в этой главе и в главе 14. Например, `value_type` каждого контейнера всегда представляет тип его элементов.

[1] Обзор класса valarray и его математических возможностей вы найдете в документации (https://en.cppreference.com/w/cpp/numeric/valarray) и в статье «std::valarray Class in C++» (https://www.geeksforgeeks.org/std-valarray-class-c).

Вложенный тип	Описание
`allocator_type`	Тип объекта, который является распределителем памяти для контейнера. Отсутствует в контейнере `array`. Каждый контейнер, использующий распределитель памяти, предоставляет распределитель по умолчанию, подходящий большинству программистов. Создание пользовательских распределителей выходит за рамки этой книги
`value_type`	Тип элементов контейнера
`reference`	Тип для объявления ссылки на элемент контейнера
`const_reference`	Тип для объявления ссылки на элемент константного контейнера
`pointer`	Тип для объявления указателя на элемент контейнера
`const_pointer`	Тип для объявления указателя на элемент константного контейнера
`iterator`	Итератор, указывающий на элемент контейнера
`const_iterator`	Итератор, указывающий на элемент константного контейнера. Только для чтения элементов и для операций с константами
`reverse_iterator`	Обратный (реверсивный) итератор, указывающий на элемент контейнера. Для перебора элементов контейнера в обратном порядке (от конца к началу). Не предоставляется контейнером `forward_list`
`const_reverse_iterator`	Обратный (реверсивный) итератор, указывающий на элемент константного контейнера. Только для чтения элементов и для операций с константами. Для перебора элементов контейнера в обратном порядке (от конца к началу). Не предоставляется контейнером `forward_list`
`difference_type`	Тип, представляющий количество элементов между двумя итераторами, указывающими на элементы одного и того же контейнера. Значение этого типа возвращается перегруженным оператором «минус» (-), если он определен
`size_type`	Тип для подсчета элементов в контейнере и для индексирования последовательного контейнера с произвольным доступом

13.2.2. Общие функции контейнеров

Большинство контейнеров предоставляет вам аналогичную функциональность. Многие операции применимы ко всем контейнерам, другие — только к некоторым категориям контейнеров. В следующих таблицах перечислены функции, доступные в большинстве контейнеров стандартной библиотеки. Прежде чем использовать какой-либо контейнер, вы должны изучить его возможности. Полный список всех функций, предназначенных для работы с каждым контейнером,

вы найдете на сайте cppreference.com[1]. Несколько функций, не упомянутых в этом разделе, рассматриваются на протяжении всей главы, по ходу демонстрации соответствующих последовательных контейнеров, ассоциативных контейнеров и адаптеров контейнеров.

Специальные функции контейнеров

В следующей таблице перечислены специальные функции, предоставляемые каждым контейнером. Кроме того, каждый класс контейнера обычно предоставляет множество перегруженных конструкторов для инициализации контейнеров и адаптеров контейнеров разными способами. Например, каждый последовательный и ассоциативный контейнер может быть инициализирован из `initializer_list`.

Специальная функция контейнера	Описание
Конструктор по умолчанию	Инициализирует пустой контейнер
Копирующий конструктор	Инициализирует контейнер копией существующего контейнера того же типа
Оператор копирующего присваивания	Копирует элементы одного контейнера в другой
Перемещающий конструктор	Перемещает содержимое существующего контейнера в новый контейнер того же типа. После этой операции исходный контейнер не содержит данных. Это ускоряет работу программы по сравнению с копированием каждого элемента существующего контейнера
Оператор перемещающего присваивания	Перемещает содержимое одного контейнера в другой контейнер того же типа. После этой операции исходный контейнер не содержит данных. Это ускоряет работу программы по сравнению с копированием каждого элемента существующего контейнера
Деструктор	Уничтожает все элементы контейнера

Операторы равенства и сравнения

Операторы <, <=, >, >=, == и != поддерживаются большинством контейнеров. В C++17 и более ранних версиях они перегружены как свободные функции. В C++20 операторы <, <=, >, >= и != генерируются компилятором на основе нового оператора трехстороннего сравнения <=> и оператора ==. Как было показано в разделе 11.7, компилятор C++ может использовать оператор <=> для реализации любой операции

[1] «Container Library — Member Functions Table». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/container.

равенства или сравнения, переопределяя ее в соответствии с синтаксисом этого оператора. Например, компилятор переопределяет `x < y` таким образом:

```
(x <=> y) < 0
```

Неупорядоченные ассоциативные контейнеры не поддерживают операторы `<`, `<=`, `>` и `>=`. Адаптер `priority_queue` не поддерживает операторы равенства и сравнения.

Функции контейнеров, возвращающие итераторы

В следующей таблице перечислены функции классов контейнеров, возвращающие итераторы. Контейнер `forward_list` не содержит функции `rbegin`, `rend`, `crbegin` и `crend`.

Функция контейнера	Описание
`begin`	Возвращает `iterator` или `const_iterator` (в зависимости от того, является ли контейнер константным), указывающий на первый элемент контейнера
`end`	Возвращает `iterator` или `const_iterator` (в зависимости от того, является ли контейнер константным), указывающий на следующую позицию после конца контейнера
`cbegin` (C++11)	Возвращает `const_iterator`, указывающий на первый элемент контейнера
`cend` (C++11)	Возвращает `const_iterator`, указывающий на следующую позицию после конца контейнера
`rbegin`	Возвращает `reverse_iterator` или `const_reverse_iterator` (в зависимости от того, является ли контейнер константным), указывающий на последний элемент контейнера
`rend`	Возвращает `reverse_iterator` или `const_reverse_iterator` (в зависимости от того, является ли контейнер константным), указывающий на позицию перед первым элементом контейнера
`crbegin` (C++11)	Возвращает `const_reverse_iterator`, указывающий на последний элемент контейнера
`crend` (C++11)	Возвращает `const_reverse_iterator`, указывающий на позицию перед первым элементом контейнера

Другие функции контейнеров

В следующей таблице перечислены различные дополнительные функции контейнеров. Если функция поддерживается не всеми контейнерами, мы указываем это в таблице. Полный список функций каждого контейнера приведен на странице документации к нему, которую вы найдете здесь:

https://en.cppreference.com/w/cpp/container

Функция контейнера	Описание
`clear`	Удаляет все элементы из контейнера. Не поддерживается контейнером `array`
`contains` (C++20)	Возвращает `true`, если указанный ключ есть в контейнере; в противном случае возвращает `false`. Поддерживается только ассоциативными контейнерами
`empty`	Возвращает `true`, если контейнер пуст; в противном случае возвращает `false`
`emplace` (C++11)	Добавляет элемент в указанную позицию в контейнере. Если в ней уже есть элемент, создает отдельный объект и перемещает его в эту позицию. Не поддерживается контейнером `array`. Контейнер `forward_list` поддерживает функции `emplace_after` и `emplace_front`, но не `emplace`. Ассоциативные контейнеры поддерживают функции `emplace` и `emplace_hint`
`erase`	Удаляет один или несколько элементов из контейнера. Не поддерживается контейнером `array`. Контейнер `forward_list` поддерживает функцию `erase_after`, но не `erase`
`extract` (C++17)	Ассоциативные контейнеры — это упорядоченные структуры данных, состоящие из узлов, содержащих значения. Функция ассоциативного контейнера `extract` удаляет узел из контейнера и возвращает объект `node_type` этого контейнера
`insert`	Вставляет элемент в контейнер. Функция перегружена для поддержки семантики копирования и перемещения. Не поддерживается контейнером `array`. Контейнер `forward_list` поддерживает функцию `insert_after`, но не `insert`
`max_size`	Возвращает максимальное количество элементов контейнера
`size`	Возвращает количество элементов, находящихся в данный момент в контейнере. Не поддерживается контейнером `forward_list`
`swap`	Меняет местами содержимое двух контейнеров

13.2.3. Требования к элементам контейнера

Перед использованием контейнера стандартной библиотеки важно убедиться, что тип помещаемых в него элементов поддерживает минимально необходимую функциональность. Например:

- Если для вставки элемента в контейнер требуется копирование объекта, тип этого объекта должен предоставлять копирующий конструктор и оператор копирующего присваивания.
- Если для вставки элемента в контейнер требуется перемещение объекта, тип этого объекта должен предоставлять перемещающий конструктор и оператор

перемещающего присваивания (семантика перемещения рассматривалась в главе 11).

- Упорядоченные ассоциативные контейнеры и многие алгоритмы требуют, чтобы элементы были сравнимыми, поэтому соответствующий тип объектов должен поддерживать операции сравнения.

Когда вы просматриваете документацию контейнера, непосредственно в документации стандарта C++ или на сторонних сайтах, например cppreference.com, вы можете заметить различные **именованные требования (named requirements)**, например `CopyConstructible`, `MoveAssignable` или `EqualityComparable`. Они помогут вам определить совместимость вашего типа данных с различными контейнерами стандартной библиотеки и алгоритмами C++. Здесь вы можете просмотреть список именованных требований и их описания:

https://en.cppreference.com/w/cpp/named_req

В C++20 многие именованные требования формализованы в виде концептов, которые мы рассмотрим в главах 14 и 15.

13.3. Работа с итераторами

Итераторы имеют много общего с указателями. Они указывают на элементы в последовательных контейнерах и ассоциативных контейнерах, но они также содержат информацию о состоянии контейнеров, с которыми работают. Итераторы реализованы для всех типов контейнеров. Синтаксис операций с итераторами унифицирован для всех контейнеров. Например, применение оператора * к итератору разыменовывает его, чтобы вы могли получить доступ к элементу, на который он указывает. Применение оператора ++ к итератору перемещает его к следующему элементу контейнера. Эта унификация является ключевым аспектом итераторов и позволяет алгоритмам стандартной библиотеки работать с различными типами контейнеров, используя динамический полиморфизм.

Последовательные контейнеры и ассоциативные контейнеры предоставляют функции `begin` и `end`. Функция `begin` возвращает итератор, указывающий на первый элемент контейнера. Функция `end` возвращает итератор, указывающий на **ограничитель (sentinel)** — несуществующий элемент, следующий за последним элементом контейнера и используемый как индикатор конца контейнера. Вы часто будете использовать ограничитель в операциях сравнения, проверяя, достиг ли конца контейнера инкрементируемый итератор.

13.3.1. Использование istream_iterator для ввода и ostream_iterator для вывода

Мы используем итераторы для работы с **последовательностями (sequences)**, которые также называются **диапазонами (ranges)**. Они могут находиться в контейнерах, но они также могут быть **последовательностями ввода (input**

sequences) и **последовательностями вывода** (**output sequences**). Код на илл. 13.1 демонстрирует:

- получение информации из стандартного потока ввода (ввод последовательности данных в программу) с помощью `istream_iterator`;
- отправку информации в стандартный поток вывода (вывод последовательности данных из программы) с помощью `ostream_iterator`.

Программа получает от пользователя два целых числа и выводит их сумму. Как вы увидите далее в этой главе, `istream_iterator` и `ostream_iterator` можно использовать с алгоритмами стандартной библиотеки для создания мощных инструкций. Например, в последующих примерах будет использоваться `ostream_iterator` в сочетании с алгоритмом `copy` для копирования элементов контейнера в стандартный поток вывода одной инструкцией.

```
1 // fig13_01.cpp
2 // Демонстрация ввода и вывода с помощью итераторов
3 #include <iostream>
4 #include <iterator> // ostream_iterator и istream_iterator
5
6 int main() {
7    std::cout << "Enter two integers: ";
8
9    // Создаем istream_iterator для ввода значений int из cin
10   std::istream_iterator<int> inputInt{std::cin};
11
12   const int number1{*inputInt}; // Получаем int из стандартного потока ввода
13   ++inputInt; // Перемещаем итератор для ввода следующего значения
14   const int number2{*inputInt}; // Получаем int из стандартного потока ввода
15
16   // Создаем ostream_iterator для вывода значений int в cout
17   std::ostream_iterator<int> outputInt{std::cout};
18
19   std::cout << "The sum is: ";
20   *outputInt = number1 + number2; // Выводим результат в cout
21   std::cout << "\n";
22 }
```

```
Enter two integers: 12 25
The sum is: 37
```

Илл. 13.1 | Демонстрация ввода и вывода с помощью итераторов

istream_iterator

В строке 10 создается `istream_iterator`, способный вводить числа `int` из стандартного объекта ввода `cin`. В строке 12 итератор `inputInt` разыменовывается для ввода из `cin` первого целого числа, которым инициализируется `number1`. Оператор разыменования `*`, примененный к `inputInt`, получает значение из потока. В строке 13 итератор `inputInt` перемещается к следующему значению из

потока ввода. В строке 14 из `inputInt` вводится следующее число `int`, которым инициализируется `number2`. Здесь можно использовать либо префиксный, либо постфиксный инкремент. Мы используем префиксную версию по соображениям быстродействия, поскольку она не создает временный объект.

ostream_iterator

В строке 17 создается `ostream_iterator`, способный выводить числа `int` в стандартный объект вывода `cout`. В строке 20 разыменованному итератору (`*outputInt`) присваивается сумма чисел `number1` и `number2`, при этом результат выводится в `cout`.

13.3.2. Категории итераторов

В следующей таблице перечислены все категории итераторов. Каждая из них имеет определенный набор функциональных возможностей. На протяжении всей главы мы обсуждаем, какую категорию итераторов поддерживает каждый из контейнеров. В главе 14 вы увидите, что минимальные требования алгоритмов к итераторам определяют, какие контейнеры совместимы с каждым из алгоритмов.

Категория итератора	Описание
`input` (итератор ввода)	Используется для чтения элементов из контейнера. Может перемещаться только вперед (от начала к концу контейнера), по одному элементу за раз. Поддерживает только однопроходные алгоритмы. Один и тот же итератор ввода нельзя использовать для повторного перебора последовательности
`output` (итератор вывода)	Используется для записи элементов в контейнер. Может перемещаться только вперед, по одному элементу за раз. Поддерживает только однопроходные алгоритмы. Один и тот же итератор вывода нельзя использовать для повторного перебора последовательности. Итераторы последующих категорий в этой таблице тоже можно использовать для записи в контейнер, если они указывают на неконстантные данные
`forward` (однонаправленный итератор)	Обладает возможностями итераторов ввода и вывода, а также сохраняет свою позицию в контейнере. Может использоваться для многократного перебора последовательности, поэтому поддерживает многопроходные алгоритмы
`bidirectional` (двунаправленный итератор)	Обладает возможностями однонаправленного итератора, но с дополнительной возможностью перемещаться назад (от конца к началу контейнера). Поддерживает многопроходные алгоритмы

Категория итератора	Описание
random access (итератор произвольного доступа)	Обладает возможностями двунаправленного итератора, а также возможностью прямого доступа к любому элементу контейнера (то есть перехода вперед или назад на любое количество элементов). Может участвовать в операциях сравнения реляционными операторами
contiguous (непрерывный итератор)	Итератор произвольного доступа, требующий, чтобы все элементы были расположены в смежных ячейках памяти. Эту категорию представили в C++17, но формализовали только в C++20

13.3.3. Поддерживаемые контейнеры

Категория итераторов, которую поддерживает каждый контейнер, определяет возможность использования этого контейнера с соответствующими алгоритмами. Контейнеры, поддерживающие итераторы произвольного доступа, могут использоваться со всеми алгоритмами стандартной библиотеки. Указатели в традиционных массивах могут использоваться как итераторы. В следующей таблице показаны категории итераторов каждого контейнера. Последовательные контейнеры, ассоциативные контейнеры, строки класса `string` и традиционные массивы — со всем этим можно работать при помощи итераторов.

Контейнер	Итераторы	Контейнер	Итераторы
Последовательные контейнеры		**Неупорядоченные ассоциативные контейнеры**	
`vector`	непрерывные	`unordered_set`	однонаправленные
`array`	непрерывные	`unordered_multiset`	однонаправленные
`deque`	произвольного доступа	`unordered_map`	однонаправленные
`list`	двунаправленные	`unordered_multimap`	однонаправленные
`forward_list`	однонаправленные		
Упорядоченные ассоциативные контейнеры		**Адаптеры контейнеров**	
`set`	двунаправленные	`stack`	нет
`multiset`	двунаправленные	`queue`	нет
`map`	двунаправленные	`priority_queue`	нет
`multimap`	двунаправленные		

13.3.4. Предопределенные типы итераторов

В следующей таблице показаны предопределенные типы итераторов, которые находятся в определениях классов контейнеров стандартной библиотеки. Не каждый тип итераторов определен для каждого контейнера. Константные итераторы предназначены для работы с контейнерами, которые не должны изменяться. Обратные итераторы, также называемые **реверсивными (reverse)**, перемещаются по контейнеру назад: от конца к началу.

Предопределенный тип итератора	Перемещается при инкременте (++)	Возможности
`iterator`	вперед	чтение и запись
`const_iterator`	вперед	чтение
`reverse_iterator`	назад	чтение и запись
`const_reverse_iterator`	назад	чтение

Операции, выполняемые с `const_iterator`, возвращают константные ссылки для предотвращения изменения элементов контейнера. Использование `const_iterator`, когда это уместно, является еще одним примером принципа наименьших привилегий.

13.3.5. Операторы для работы с итераторами

В следующей таблице перечислены операторы и совместимые с ними типы итераторов. Кроме указанных операторов, итераторы должны предоставлять конструкторы по умолчанию (однонаправленные итераторы и выше), копирующие конструкторы и операторы копирующего присваивания. Однонаправленные итераторы поддерживают оператор ++ и все возможности итераторов ввода и вывода. Двунаправленные итераторы поддерживают оператор - - и все возможности однонаправленных итераторов. Итераторы произвольного доступа и непрерывные итераторы поддерживают все операции, показанные в таблице. Итератор ввода или вывода невозможно сохранить и впоследствии использовать сохраненное значение.

Операция с итератором	Описание
Все итераторы	
`++p`	Выполняет префиксный инкремент итератора
`p++`	Выполняет постфиксный инкремент итератора
`p = p1`	Присваивает один итератор другому
Итераторы ввода	
`*p`	Разыменовывает итератор как значение `const` *lvalue*
`p->m`	Использует итератор для чтения элемента `m`

Операция с итератором	Описание
`p == p1`	Сравнивает итераторы, проверяя их равенство
`p != p1`	Сравнивает итераторы, проверяя их неравенство
Итераторы вывода	
`*p`	Разыменовывает итератор как значение *lvalue*
`p = p1`	Присваивает один итератор другому
Однонаправленные итераторы	Обеспечивают всю функциональность итераторов ввода и вывода
Двунаправленные итераторы	
`--p`	Выполняет префиксный декремент итератора
`p--`	Выполняет постфиксный декремент итератора
Итераторы произвольного доступа	
`p += i`	Перемещает итератор `p` вперед (выполняет инкремент) на `i` позиций
`p -= i`	Перемещает итератор `p` назад (выполняет декремент) на `i` позиций
`p + i` или `i + p`	Значение выражения — итератор, расположенный на позиции `p + i`
`p - i`	Значение выражения — итератор, расположенный на позиции `p - i`
`p - p1`	Значение выражения — целое число, обозначающее расстояние (то есть количество элементов) между двумя элементами в одном контейнере
`p[i]`	Возвращает ссылку на элемент, смещенный от `p` на `i` позиций вперед
`p < p1`	Возвращает `true`, если итератор `p` меньше итератора `p1` (то есть если `p` находится ближе к началу контейнера, чем `p1`); в противном случае возвращает `false`
`p <= p1`	Возвращает `true`, если итератор `p` меньше или равен итератору `p1` (то есть если `p` находится не дальше от начала контейнера, чем `p1`); в противном случае возвращает `false`
`p > p1`	Возвращает `true`, если итератор `p` больше итератора `p1` (то есть если `p` находится дальше от начала контейнера, чем `p1`); в противном случае возвращает `false`
`p >= p1`	Возвращает `true`, если итератор `p` больше или равен итератору `p1` (то есть если `p` находится не ближе к началу контейнера, чем `p1`); в противном случае возвращает `false`

13.4. Несколько слов об алгоритмах для обработки контейнеров

Стандартная библиотека предоставляет множество алгоритмов, которые вы можете использовать для обработки разных контейнеров. Вставка, удаление, поиск, сортировка и другие действия применимы к некоторым или ко всем последовательным и ассоциативным контейнерам. Алгоритмы работают с элементами контейнеров косвенно: через итераторы. Многие алгоритмы работают с последовательностями, которые заданы итераторами, указывающими на первый элемент последовательности и на элемент, следующий за последним элементом, — это называется **полуоткрытым диапазоном (half-open range)**. И уже многие алгоритмы поддерживают диапазоны C++20. Вы можете создавать собственные алгоритмы, обрабатывающие контейнеры аналогичным образом, чтобы их можно было использовать с контейнерами и итераторами стандартной библиотеки. Во многих примерах этой главы алгоритм `copy` копирует содержимое контейнера в стандартный поток вывода. Мы рассмотрим целый ряд алгоритмов стандартной библиотеки в главе 14.

13.5. Последовательные контейнеры

Стандартная библиотека C++ предоставляет пять последовательных контейнеров: `array`, `vector`, `deque`, `list` и `forward_list`. Контейнеры `array`, `vector` и `deque` основаны на традиционных массивах. Контейнеры `list` и `forward_list` реализуют структуры данных связных списков. В главе 6 мы много работали с контейнером `array`, поэтому в этой главе уже не будем его обсуждать. В главе 6 мы также начали работать с контейнером `vector`, который теперь рассмотрим более подробно.

Быстродействие и выбор контейнера

В разделе 13.2.2 были рассмотрены операции, поддерживаемые большинством контейнеров стандартной библиотеки. Помимо этих операций, каждый контейнер может предоставлять дополнительные возможности. Многие из них являются общими для нескольких контейнеров, но они не всегда одинаково эффективны. В C++ Core Guidelines утверждается, что контейнер `vector` подходит для большинства приложений[1]. Однако вам следует убедиться, что выбранный вами контейнер соответствует вашей конкретной задаче и вашим требованиям к быстродействию.

Вставка элементов в конец контейнера `vector` очень эффективна. При необходимости `vector` увеличивается, чтобы вместить новый элемент. Но вставка и удаление элементов в середине контейнера `vector` менее эффективны. Каждый элемент,

[1] C++ Core Guidelines, «SL.con.2: Prefer Using STL vector By Default Unless You Have a Reason to Use a Different Container». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rsl-vector.

следующий за точкой вставки или удаления, приходится перемещать, поскольку элементы контейнера `vector` занимают смежные ячейки в памяти.

Для приложений, часто вставляющих или удаляющих элементы на обеих границах контейнера, предпочтительнее не `vector`, а `deque`. Класс `deque` обеспечивает вычислительную сложность $O(1)$ для операций вставки и удаления элементов в начале контейнера[1], что делает его более эффективным, чем `vector`, обеспечивающий для таких операций вычислительную сложность $O(n)$[2]. Если вы не знакомы с нотацией «*O*-большое», прочтите в разделе 13.12 краткое введение в эту тему.

Приложения, часто вставляющие или удаляющие элементы в середине и (или) на границах контейнеров, иногда используют контейнер `list`, потому что операции вставки и удаления элементов в любом месте структуры данных реализованы в нем очень эффективно.

13.6. Последовательный контейнер vector

Контейнер `vector` (см. раздел 6.15) представляет собой динамическую структуру данных со смежными ячейками памяти. Он обеспечивает быстрый индексированный доступ с помощью перегруженного оператора индекса `[]`, подобно массиву в стиле C или объекту `array`. Выбирайте `vector` для наилучшего быстродействия при произвольном доступе к контейнеру, который может расширяться. Когда емкость контейнера `vector` превышена, он:

- создает традиционный массив большего размера;
- копирует или перемещает (в зависимости от того, что поддерживает тип элементов) все исходные элементы в созданный традиционный массив;
- освобождает прежний традиционный массив.

13.6.1. Использование векторов и итераторов

На илл. 13.2 показано несколько функций класса `vector`, многие из которых доступны в каждом последовательном и ассоциативном контейнере. Когда мы добавляем элементы в `vector<int>` в этом примере, мы вызываем нашу функцию `showResult` (строки 9–12), чтобы вывести на экран новые значения:

- размер (`size`) — количество элементов, содержащихся в векторе в данный момент;
- емкость (`capacity`) — количество элементов, которые вектор может хранить, прежде чем ему потребуется динамическое расширение для размещения большего количества элементов.

---

[1] «std::deque». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/container/deque.

[2] «std::vector». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/container/vector.

```
// fig13_02.cpp
// Демонстрация шаблона класса vector стандартной библиотеки
#include <fmt/format.h> // Или в C++20: #include <format>
#include <iostream>
#include <ranges>
#include <vector> // Определение шаблона класса vector

// Отображаем добавленное значение, обновленный размер и емкость
void showResult(int value, size_t size, size_t capacity) {
   std::cout << fmt::format("appended: {}; size: {}; capacity: {}\n",
      value, size, capacity);
}

```

Илл. 13.2 | Демонстрация шаблона класса vector стандартной библиотеки

Создание вектора, отображение его исходного размера и емкости

В строке 15 создается объект `vector<int>`. Конструктор по умолчанию создает пустой вектор (размер и емкость равны `0`), поэтому память для этого объекта будет выделяться только при добавлении элементов. В строках 17–18 отображаются размер и емкость. Функция `size` возвращает количество элементов, хранящихся в контейнере в данный момент[1]. Функция `capacity` возвращает текущую емкость вектора.

```
int main() {
   std::vector<int> integers{}; // Создаем вектор элементов int

   std::cout << "Size of integers: " << integers.size()
      << "\nCapacity of integers: " << integers.capacity() << "\n\n";

```

```
Size of integers: 0
Capacity of integers: 0
```

Функция push_back

В строках 21–24 вызывается функция `push_back` для добавления одного элемента в вектор. В каждой итерации цикла вызывается функция `showResult`, отображающая добавленный элемент, новый размер и новую емкость. Функция `push_back` предоставляется всеми последовательными контейнерами, кроме `array` и `forward_list`. Все последовательные контейнеры, кроме `array` и `forward_list`, также предоставляют функцию `push_front`. Если размер вектора равен его емкости при добавлении нового элемента, вектор увеличивает свою емкость, чтобы вмещать больше элементов.

[1] Контейнер forward_list не предоставляет функцию size.

```
20    // Добавляем 1-10 к integers, отображаем размер и емкость
21    for (int i : std::views::iota(1, 11)) {
22       integers.push_back(i); // Функция push_back есть в vector, deque и list
23       showResult(i, integers.size(), integers.capacity());
24    }
25
```

```
appended: 1; size: 1; capacity: 1
appended: 2; size: 2; capacity: 2
appended: 3; size: 3; capacity: 3
appended: 4; size: 4; capacity: 4
appended: 5; size: 5; capacity: 6
appended: 6; size: 6; capacity: 6
appended: 7; size: 7; capacity: 9
appended: 8; size: 8; capacity: 9
appended: 9; size: 9; capacity: 9
appended: 10; size: 10; capacity: 13
```

Изменения размера и емкости после добавления элементов

В стандарте C++ не указан конкретный способ увеличения вектора для размещения большего количества элементов, но в любом случае это длительная операция. Некоторые реализации C++ удваивают пропускную способность вектора, а некоторые увеличивают емкость только в полтора раза, как показано в выходных данных Visual C++. Это становится очевидным с того момента, когда размер и емкость равны 4:

- после добавления элемента 5 размер увеличился до 5, а емкость — с 4 до 6, то есть осталось место еще для одного элемента;
- после добавления элемента 6 размер увеличился до 6, а емкость осталась равной 6;
- после добавления элемента 7 размер увеличился до 7, а емкость — с 6 до 9, то есть осталось место еще для двух элементов;
- после добавления элемента 8 размер увеличился до 8, а емкость осталась равной 9;
- после добавления элемента 9 размер увеличился до 9, а емкость осталась равной 9;
- после добавления элемента 10 размер увеличился до 10, а емкость — с 9 до 13 (примерно в полтора раза с округлением до ближайшего целого числа), то есть осталось место еще для трех элементов до следующего выделения памяти.

Увеличение емкости вектора в g++

Разработчики библиотеки C++ используют различные схемы для минимизации нагрузки на компьютер при изменении размера вектора, поэтому выходные

данные этой программы могут варьироваться в зависимости от того, как реализовано расширение емкости в вашем компиляторе. Например, GNU g++ удваивает емкость вектора, когда требуется больше места, генерируя такие выходные данные:

```
appended: 1; size: 1; capacity: 1
appended: 2; size: 2; capacity: 2
appended: 3; size: 3; capacity: 4
appended: 4; size: 4; capacity: 4
appended: 5; size: 5; capacity: 8
appended: 6; size: 6; capacity: 8
appended: 7; size: 7; capacity: 8
appended: 8; size: 8; capacity: 8
appended: 9; size: 9; capacity: 16
appended: 10; size: 10; capacity: 16
```

Проблемы изменения емкости вектора

Некоторые программисты изначально создают вектор с большой емкостью. В этом случае, если вектор хранит небольшое количество элементов, он будет занимать лишнее место. Однако это может значительно повысить быстродействие, если программа должна добавлять в вектор большое количество элементов и перераспределять память для размещения добавленных элементов. Это классический компромисс между пространством и временем. Разработчики библиотек должны находить баланс между объемом занимаемой памяти и временем, необходимым для выполнения различных операций с векторами.

Удвоение емкости вектора в тех случаях, когда надо добавить всего один элемент, может быть нерациональным. Например, если реализация удваивает емкость при каждом расширении, то заполненный вектор, содержащий 1 000 000 элементов, при добавлении следующего элемента увеличит емкость в расчете на 2 000 000 элементов. Таким образом, память для 999 999 элементов будет занята напрасно. Для точного управления выделением памяти вы можете использовать функцию `reserve` класса `vector`.

Вывод содержимого вектора с помощью итераторов

В строках 28–31 содержимое вектора выводится на экран. В строке 28 инициализируется управляющая переменная `constIterator` с помощью функции `cbegin` класса `vector`, которая возвращает `const_iterator`, указывающий на первый элемент вектора. Мы используем ключевое слово `auto` для автоматического определения типа управляющей переменной:

```
vector<int>::const_iterator
```

Это упрощает код и снижает количество ошибок при работе со сложными типами.

```
26    std::cout << "\nOutput integers using iterators: ";
27
28    for (auto constIterator{integers.cbegin()};
```

```
29       constIterator != integers.cend(); ++constIterator) {
30       std::cout << *constIterator << ' ';
31    }
32
```

```
Output integers using iterators: 1 2 3 4 5 6 7 8 9 10
```

Цикл продолжается, пока `constIterator` не достигнет конца вектора. Это определяется сравнением `constIterator` с результатом вызова функции `cend` вектора. Когда переменная `constIterator` будет равна этому значению, цикл завершится. Попытка разыменования итератора, указывающего на позицию вне своего контейнера, является логической ошибкой. Итератор, возвращаемый функциями `end` или `cend`, никогда не должен разыменовываться или инкрементироваться.

В строке 30 разыменовывается `constIterator` для получения значения текущего элемента. Помните, что итератор работает как указатель на элемент, а оператор * перегружен для возврата ссылки на элемент. Выражение `++constIterator` (строка 29) перемещает итератор на следующий элемент вектора. Вы могли бы заменить этот цикл более простым, используя инструкцию `for` для диапазона значений:

```
for (auto const& item : integers) {
   std::cout << item << ' ';
}
```

Инструкция `for` для диапазона значений использует итераторы «под капотом» компилятора.

Вывод содержимого вектора в обратном порядке с помощью const_reverse_iterator

В строках 36–39 элементы вектора выводятся в обратном порядке. Функции `crbegin` и `crend` возвращают итераторы `const_reverse_iterator`, представляющие начальную и конечную точки для перебора элементов контейнера в обратном порядке. Большинство последовательных контейнеров и упорядоченных ассоциативных контейнеров поддерживают реверсивную итерацию. Класс `vector` также предоставляет функции `rbegin` и `rend` для получения неконстантных итераторов `reverse_iterator`.

```
33    std::cout << "\nOutput integers in reverse using iterators: ";
34
35    // Выводим элементы в обратном порядке с помощью const_reverse_iterator
36    for (auto reverseIterator{integers.crbegin()};
37       reverseIterator != integers.crend(); ++reverseIterator) {
38       std::cout << *reverseIterator << ' ';
39    }
40
41    std::cout << "\n";
42 }
```

```
Output integers in reverse using iterators: 10 9 8 7 6 5 4 3 2 1
```

Функция shrink_to_fit (C++11)

Начиная с C++11, вы можете вызывать функцию `shrink_to_fit`, которая отправляет запрос на возвращение в систему неиспользуемой памяти контейнера `vector` или `deque` путем уменьшения его емкости до текущего количества элементов. Впрочем, согласно стандарту C++, компиляторы могут игнорировать такие запросы, чтобы оптимизировать исполняемый код; это зависит от конкретной реализации компилятора.

13.6.2. Функции для обработки элементов вектора

На илл. 13.3 показаны вызовы функций для извлечения и обработки элементов вектора. В строке 12 `vector<int>` инициализируется скобочной инициализацией. В данном случае мы объявили тип вектора как `std::vector`. С помощью CTAD компилятор определяет типы элементов по значениям `int` из списка инициализаторов. В строке 13 используется конструктор класса `vector`, принимающий в качестве аргументов два итератора для инициализации вектора `integers` копией элементов вектора `values` в диапазоне от `values.begin()` до ограничителя (но не включая его) `values.cend()`.

```
 1 // fig13_03.cpp
 2 // Демонстрация функций шаблона класса vector стандартной библиотеки,
 3 // предназначенных для обработки элементов
 4 #include <algorithm> // Алгоритм copy
 5 #include <fmt/format.h> // Или в C++20: #include <format>
 6 #include <iostream>
 7 #include <ranges>
 8 #include <iterator> // Итератор ostream_iterator
 9 #include <vector>
10
11 int main() {
12    std::vector values{1, 2, 3, 4, 5}; // Используем CTAD
13    std::vector<int> integers{values.cbegin(), values.cend()};
14    std::ostream_iterator<int> output{std::cout, " "};
15
```

Илл. 13.3 | Демонстрация функций для обработки элементов вектора

Итератор ostream_iterator

В строке 14 определяется итератор `ostream_iterator` с именем `output`, который может использоваться для вывода целых чисел, разделенных пробелом, через `cout`. Итератор `ostream_iterator<int>` выводит только числа типа `int`. Первый аргумент конструктора определяет выходной поток, а второй аргумент является строкой, задающей разделитель для выходных значений, — в нашем случае это символ пробела. В этом примере мы используем `ostream_iterator` (заголовок `<iterator>`) для вывода содержимого вектора.

Алгоритм copy

В строке 17 алгоритм `copy` стандартной библиотеки (заголовок `<algorithm>`) используется для копирования содержимого вектора `integers` в стандартный поток вывода. Эта версия алгоритма принимает три аргумента. Первые два — итераторы, указывающие на элементы для копирования из вектора `integers`: от `integers.cbegin()` до ограничителя (но не включая его) `integers.cend()`. Аргументы должны соответствовать требованиям итератора ввода, например итератора `const_iterator`, позволяющего считывать значения из контейнера. Они также должны ссылаться на один и тот же контейнер таким образом, чтобы многократное применение оператора `++` к первому итератору в конечном итоге приводило его ко второму итератору. Третий аргумент указывает, куда надо копировать элементы. Это должен быть итератор вывода, с помощью которого значение может быть сохранено или выведено. Итератор вывода — это `ostream_iterator`, подключенный к `cout`, поэтому в строке 17 элементы копируются в стандартный поток вывода.

```
16    std::cout << "integers contains: ";
17    std::copy(integers.cbegin(), integers.cend(), output);
18
```

```
integers contains: 1 2 3 4 5
```

Обычные диапазоны

До C++20 «диапазон» элементов контейнера описывался итераторами, указывающими начальную и конечную позиции, чаще всего через вызовы функций `begin` и `end` (или аналогичных). Начиная с C++20, стандарт C++ называет такие диапазоны **обычными диапазонами (common ranges)**, поэтому их не следует путать с **диапазонами (ranges)**, введенными в C++20. С этого момента мы будем использовать термин «обычный диапазон» при упоминании диапазонов, определяемых двумя итераторами, и термин «диапазон» — при упоминании диапазонов библиотеки `ranges` C++20.

Функции front и back

В строках 19–20 выводятся первый и последний элементы вектора с помощью функций `front` и `back`, доступных в большинстве последовательных контейнеров. Обратите внимание на разницу между функциями `front` и `begin`. Функция `front` возвращает ссылку на первый элемент вектора, а функция `begin` возвращает итератор, указывающий на первый элемент вектора. Аналогично функция `back` возвращает ссылку на последний элемент вектора, а функция `end` возвращает итератор, указывающий на ограничитель (на следующую позицию после последнего элемента). Результаты вызовов функций `front` и `back` не определены при их вызове для пустого вектора.

```
19    std::cout << fmt::format("\nfront: {}\nback: {}\n\n",
20       integers.front(), integers.back());
21
```

```
front: 1
back: 5
```

Доступ к элементам вектора

В строках 22–23 показаны два способа доступа к элементам контейнера `vector` (или `deque`). В строке 22 используется оператор индекса `[]`, возвращающий ссылку на значение в указанной позиции (или константную ссылку, в зависимости от возможности изменения контейнера). Функция `at` (строка 23) выполняет такую же операцию, но с проверкой границ. Функция `at` сначала проверяет аргумент и определяет, находится ли он в границах контейнера. Если нет, функция `at` выдает исключение `out_of_range`, как показано в разделе 6.15.

```
22    integers[0] = 7; // Присваиваем значение 7 первому элементу контейнера
23    integers.at(2) = 10; // Присваиваем значение 10 элементу на позиции 2
24
```

Функция insert класса vector

В строке 26 используется одна из версий перегруженной функции `insert`, предоставляемой каждым последовательным контейнером (кроме `array`, который имеет фиксированный размер, и `forward_list`, который содержит функцию `insert_after`). В строке 26 значение `22` вставляется на позицию элемента, указанного итератором в первом аргументе. Итератор указывает на второй элемент, поэтому `22` становится вторым элементом, а исходный второй элемент — третьим. Другие версии функции `insert` позволяют:

- вставить нескольких копий одного и того же значения, начиная с заданной позиции;
- вставить диапазон значений из другого контейнера, начиная с заданной позиции.

Версия функции `insert` в строке 26 возвращает итератор, указывающий на вставленный элемент.

```
25    // Вставляем 22 на позицию второго элемента
26    integers.insert(integers.cbegin() + 1, 22);
27
28    std::cout << "Contents of vector integers after changes: ";
29    std::ranges::copy(integers, output);
30
```

```
Contents of vector integers after changes: 7 22 2 10 4 5
```

Алгоритм copy библиотеки ranges C++20

В строке 29 алгоритм `copy` из пространства имен `std::ranges` C++20 используется для копирования элементов `integers` в стандартный поток вывода. Эта версия функции `copy` получает только диапазон для копирования и выходной итератор, указывающий, куда надо копировать элементы. Первый аргумент — это объект, представляющий диапазон элементов с итераторами, указывающими его начало и конец, — в нашем случае это `vector`. Большинство алгоритмов из заголовка `<algorithm>`, существовавших до C++20, теперь имеют версии в библиотеке `ranges` C++20, в пространстве имен `std::ranges`. Алгоритмы `std::ranges` обычно перегружены в версии, принимающей объект диапазона, и в версии, принимающей итератор и ограничитель. В терминах библиотеки `ranges` C++20 ограничителем называется объект, обозначающий достижение конца контейнера. В главе 14 демонстрируются многие алгоритмы диапазонов C++20.

Функция erase

В строках 31 и 36 вызывается функция `erase`, доступная в большинстве последовательных и ассоциативных контейнеров, за исключением контейнера `array` с фиксированным размером и контейнера `forward_list`, в котором вместо нее есть функция `erase_after`. В строке 31 удаляется элемент, находящийся на позиции, указанной в аргументе-итератора, — в данном случае на позиции первого элемента. В строке 36 удаляются элементы обычного диапазона в промежутке, указанном двумя аргументами-итераторами, — в данном случае все элементы. В строке 38 вызывается функция `empty` (определенная для всех контейнеров и адаптеров), подтверждающая отсутствие элементов в векторе.

```
31    integers.erase(integers.cbegin()); // Удаляем первый элемент
32    std::cout << "\n\nintegers after erasing first element: ";
33    std::ranges::copy(integers, output);
34
35    // Удаляем оставшиеся элементы
36    integers.erase(integers.cbegin(), integers.cend());
37    std::cout << fmt::format("\nErased all elements: integers {} empty\n",
38       integers.empty() ? "is" : "is not");
39
```

```
integers after erasing first element: 22 2 10 4 5
Erased all elements: integers is empty
```

Как правило, функция `erase` уничтожает объекты, которые она удаляет. Однако удаление элемента, являющегося указателем на динамически размещенный объект, не приводит к удалению этого объекта, что может вызвать утечку памяти. Именно поэтому вам не следует управлять динамически выделяемой памятью с помощью простых указателей. Удаление элемента `unique_ptr` (раздел 11.5) освобождает динамически выделенную память. Для сравнения, при удалении элемента `shared_ptr` выполняется декремент счетчика ссылок на динамически выделенный объект, а память освобождается только после обнуления счетчика.

Функция insert с тремя аргументами (вставка в диапазон)

В строке 41 демонстрируется версия функции `insert`, использующая второй и третий аргументы для определения обычного диапазона элементов, вставляемых в вектор (в данном случае из `values`). Помните, что последняя позиция указывает на элемент, находящийся после последнего вставляемого элемента; копирование выполняется до указанной позиции, не включая ее. Эта версия функции `insert` возвращает итератор, указывающий на первый вставленный элемент. Если ничего не было вставлено, функция возвращает свой первый аргумент.

```
40    // Вставляем элементы из вектора values
41    integers.insert(integers.cbegin(), values.cbegin(), values.cend());
42    std::cout << "\nContents of vector integers before clear: ";
43    std::ranges::copy(integers, output);
44
```

```
Contents of vector integers before clear: 1 2 3 4 5
```

Функция clear

В строке 46 вызывается функция `clear` для очистки вектора. Все последовательные и ассоциативные контейнеры, кроме `array` с фиксированным размером, предоставляют функцию `clear`. Функция не уменьшает емкость вектора.

```
45    // Очищаем вектор integers; функция clear удаляет все элементы
46    integers.clear();
47    std::cout << fmt::format("\nAfter clear, integers {} empty\n",
48       integers.empty() ? "is" : "is not");
49 }
```

```
After clear, integers is empty
```

13.7. Последовательный контейнер list

Последовательный контейнер `list` (заголовок `<list>`) позволяет выполнять операции вставки и удаления в любом месте контейнера. Контейнер `list` реализован в виде двусвязного списка[1]: каждый узел содержит указатели на предыдущий и следующий узел в контейнере. Поэтому `list` поддерживает двунаправленные итераторы, которые могут перемещаться по контейнеру как вперед, так и назад. Любой алгоритм, требующий итераторов ввода, вывода, однонаправленных или двунаправленных, может работать с контейнером `list`. Многие функции контейнера `list` обрабатывают его элементы в виде упорядоченного набора элементов. Если предполагается много операций вставки и удаления элементов в начало или в конец контейнера, рассмотрите возможность использования контейнера `deque`

[1] «std::list». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/container/list.

(раздел 13.8) или vector. Напомним, что вставка в начало контейнера реализована в deque эффективнее, чем в vector.

Контейнер forward_list

Последовательный контейнер forward_list (заголовок <forward_list>, C++11) реализован в виде односвязного списка: каждый узел содержит указатель на следующий узел в контейнере. Поэтому forward_list поддерживает однонаправленные итераторы, позволяющие перемещаться по контейнеру в одном направлении. Любой алгоритм, требующий итераторов ввода, вывода или однонаправленных, может работать с контейнером forward_list.

Функции контейнера list

В дополнение к общим функциям последовательных контейнеров, рассмотренным в разделе 13.2, контейнер list предоставляет функции splice, push_front, pop_front, emplace_front, emplace_back, remove, remove_if, unique, merge, reverse и sort. Некоторые из них — это оптимизированные для работы с контейнером list версии алгоритмов стандартной библиотеки, рассматриваемых в главе 14. Функции push_front и pop_front также поддерживаются контейнерами forward_list и deque. На илл. 13.4 показаны некоторые функции контейнера list. Многие функции, показанные на илл. 13.2–13.3, тоже можно использовать с этим контейнером. Мы сосредоточимся на новых функциях при обсуждении примеров.

```
// fig13_04.cpp
// Демонстрация шаблона класса list стандартной библиотеки
#include <algorithm> // Алгоритм copy
#include <iostream>
#include <iterator> // ostream_iterator
#include <list> // Определение шаблона класса list
#include <vector>

// Определение шаблона функции printList; использует
// ostream_iterator и алгоритм copy для вывода элементов контейнера
template <typename T>
void printList(const std::list<T>& items) {
   if (items.empty()) { // Если в контейнере нет элементов
      std::cout << "List is empty";
   }
   else {
      std::ostream_iterator<T> output{std::cout, " "};
   std::ranges::copy(items, output);
   }
}

```

Илл. 13.4 | Демонстрация шаблона класса list стандартной библиотеки

Шаблон функции `printList` (строки 11–20) проверяет, является ли `list` пустым (строка 13), и выводит сообщение, если это так. В противном случае функция `printList` использует итератор `ostream_iterator` и алгоритм `std::ranges::copy` для копирования элементов `list` в стандартный поток вывода, как показано на илл. 13.3.

Создание объекта list

В строке 23 создается контейнер `list` с именем `values`, предназначенный для целых чисел. В строках 26–27 вызывается функция `push_front` для вставки элементов в начало контейнера. Эта функция предоставляется только контейнерами `forward_list`, `list` и `deque`. В строках 28–29 вызывается функция `push_back` для добавления элементов в конец контейнера. Функция `push_back` предоставляется всеми последовательными контейнерами, кроме `array` и `forward_list`. В строках 31–32 содержимое контейнера выводится на экран.

```
22 int main() {
23    std::list<int> values{}; // Создаем контейнер list с целыми числами
24
25    // Вставляем элементы в контейнер
26    values.push_front(1);
27    values.push_front(2);
28    values.push_back(4);
29    values.push_back(3);
30
31    std::cout << "values contains: ";
32    printList(values);
33
```

```
values contains: 2 1 4 3
```

Функция sort контейнера list

В строке 34 вызывается функция `sort` для упорядочивания элементов контейнера в порядке возрастания. Другая версия функции `sort` позволяет вам передать ей **бинарную предикатную функцию (binary predicate function)**, которая принимает два аргумента (значения из `list`), сравнивает их и возвращает результат `bool`, указывающий, должен ли первый аргумент предшествовать второму в отсортированном содержимом. Иными словами, предикатная функция указывает порядок сортировки элементов. Эта версия полезна для сортировки `list` в порядке убывания, а также для выбора способа сравнения элементов при определении порядка сортировки.

```
34    values.sort(); // Сортируем values
35    std::cout << "\nvalues after sorting contains: ";
36    printList(values);
37
```

```
values after sorting contains: 1 2 3 4
```

Функция splice контейнера list

В строке 46 вызывается функция `splice` для удаления элементов из контейнера `otherValues` и для их вставки в контейнер `values` перед позицией итератора, указанной в качестве первого аргумента. Функция `splice` с тремя аргументами позволяет удалить один элемент из контейнера, указанного в качестве второго аргумента, из позиции, указанной итератором в третьем аргументе. Функция `splice` с четырьмя аргументами использует последние два аргумента в качестве обычного диапазона для удаления из контейнера, указанного во втором аргументе, и для вставки в позицию, указанную в первом аргументе. Контейнер `forward_list` предоставляет аналогичную функцию `splice_after`.

```
38    // Вставляем элементы вектора ints в otherValues
39    std::vector ints{2, 6, 4, 8};
40    std::list<int> otherValues{}; // Создаем контейнер list с целыми числами
41    otherValues.insert(otherValues.cbegin(), ints.cbegin(), ints.cend());
42    std::cout << "\nAfter insert, otherValues contains: ";
43    printList(otherValues);
44
45    // Удаляем элементы из otherValues и добавляем их в конец values
46    values.splice(values.cend(), otherValues);
47    std::cout << "\nAfter splice, values contains: ";
48    printList(values);
49
```

```
After insert, otherValues contains: 2 6 4 8
After splice, values contains: 1 2 3 4 2 6 4 8
```

Функция merge контейнера list

После вставки новых элементов в контейнер `otherValues` и сортировки контейнеров `values` и `otherValues` в строке 61 вызывается функция `merge` для удаления всех элементов из контейнера `otherValues` и для их вставки в отсортированном порядке в контейнер `values`. Перед выполнением этой операции оба контейнера должны быть отсортированы в одном и том же порядке. Другая версия `merge` позволяет вам передать ей бинарную предикатную функцию, принимающую два аргумента (значения из `list`) и возвращающую значение `bool`. Предикатная функция определяет порядок сортировки, используемый функцией `merge`, возвращая `true`, если первый аргумент предикатной функции должен быть помещен перед вторым.

```
50    values.sort(); // Сортируем values
51    std::cout << "\nAfter sort, values contains: ";
52    printList(values);
53
54    // Вставляем элементы контейнера ints в контейнер otherValues
55    otherValues.insert(otherValues.cbegin(), ints.cbegin(), ints.cend());
56    otherValues.sort(); // Сортируем otherValues
57    std::cout << "\nAfter insert and sort, otherValues contains: ";
58    printList(otherValues);
```

```
59
60    // Удаляем элементы из otherValues и вставляем в values после сортировки
61    values.merge(otherValues);
62    std::cout << "\nAfter merge:\n values contains: ";
63    printList(values);
64    std::cout << "\n otherValues contains: ";
65    printList(otherValues);
66
```

```
After sort, values contains: 1 2 2 3 4 4 6 8
After insert and sort, otherValues contains: 2 4 6 8
After merge:
   values contains: 1 2 2 2 3 4 4 4 6 6 8 8
   otherValues contains: List is empty
```

Функции pop_front и pop_back контейнера list

В строке 67 вызывается функция `pop_front` для удаления первого элемента. В строке 68 вызывается функция `pop_back` для удаления последнего элемента. Эта функция есть во всех последовательных контейнерах, кроме `array` и `forward_list`.

```
67    values.pop_front(); // Удаляем первый элемент
68    values.pop_back(); // Удаляем последний элемент
69    std::cout << "\nAfter pop_front and pop_back:\n values contains: ";
70    printList(values);
71
```

```
After pop_front and pop_back:
   values contains: 2 2 2 3 4 4 4 6 6 8
```

Функция unique контейнера list

В строке 72 вызывается функция `unique` для удаления повторяющихся соседних элементов. Если `list` отсортирован, все дубликаты удаляются. Другая версия функции `unique` позволяет вам указать предикатную функцию, которая принимает два аргумента (значения из `list`), и возвращает значение `bool`, показывающее, идентичны ли два элемента.

```
72    values.unique(); // Удаляем повторяющиеся элементы
73    std::cout << "\nAfter unique, values contains: ";
74    printList(values);
75
```

```
After unique, values contains: 2 3 4 6 8
```

Функция swap контейнера list

В строке 76 вызывается функция `swap` для обмена содержимым контейнеров `values` и `otherValues`. Эта функция есть во всех последовательных и ассоциативных контейнерах,.

```
76     values.swap(otherValues); // Меняем местами содержимое контейнеров values
                                 и otherValues
77     std::cout << "\nAfter swap:\n values contains: ";
78     printList(values);
79     std::cout << "\n otherValues contains: ";
80     printList(otherValues);
81
```

```
After swap:
   values contains: List is empty
   otherValues contains: 2 3 4 6 8
```

Функции assign и remove контейнера list

В строке 83 вызывается функция `assign` (есть во всех последовательных контейнерах) для замены содержимого контейнера `values` обычным диапазоном элементов из контейнера `otherValues`. Вторая версия функции `assign` заменяет содержимое контейнера копиями указанного во втором аргументе значения, в количестве, указанном в первом аргументе. В строке 92 вызывается функция `remove` для удаления из контейнера всех элементов со значением 4.

```
82     // Заменяем содержимое контейнера values содержимым контейнера otherValues
83     values.assign(otherValues.cbegin(), otherValues.cend());
84     std::cout << "\nAfter assign, values contains: ";
85     printList(values);
86
87     // Удаляем элементы otherValues и вставляем в values после сортировки
88     values.merge(otherValues);
89     std::cout << "\nAfter merge, values contains: ";
90     printList(values);
91
92     values.remove(4); // Удаляем все элементы со значением 4
93     std::cout << "\nAfter remove(4), values contains: ";
94     printList(values);
95     std::cout << "\n";
96 }
```

```
After assign, values contains: 2 3 4 6 8
After merge, values contains: 2 2 3 3 4 4 6 6 8 8
After remove(4), values contains: 2 2 3 3 6 6 8 8
```

13.8. Последовательный контейнер deque

Класс `deque` (заголовок `<deque>`), имя которого является сокращением от «double-ended queue», объединяет многие преимущества классов `vector` и `list`. Он обеспечивает эффективный индексированный доступ (с оператором индекса) для чтения и изменения элементов подобно классам `vector` и `array`. Он также обеспечивает эффективную вставку и удаление в начале и в конце подобно классу `list`. Класс `deque` поддерживает итераторы произвольного доступа, поэтому его можно

использовать со всеми алгоритмами стандартной библиотеки. Обычно deque используется для поддержки очереди элементов FIFO («первым пришел, первым ушел»). Фактически контейнер deque является контейнером по умолчанию для адаптера queue[1] (раздел 13.10.2).

Дополнительное место для контейнера deque может быть выделено на любом его конце. Обычно это реализуется в виде традиционных массивов указателей на выделяемые блоки памяти[2]. Из-за несмежного расположения в памяти итераторы контейнера должны быть «умнее», чем итераторы для работы с контейнерами vector и array. Контейнер deque предоставляет все те же операции, что и vector, но подобно контейнеру list, содержит функции push_front и pop_front для эффективной вставки и удаления элементов в начале контейнера.

На илл. 13.5 демонстрируются некоторые возможности контейнера deque. Многие функции, представленные на илл. 13.2–13.4, также можно использовать с этим контейнером.

```
// fig13_05.cpp
// Шаблон класса deque стандартной библиотеки
#include <algorithm> // Алгоритм copy
#include <deque> // Определение шаблона класса deque
#include <iostream>
#include <iterator> // ostream_iterator

int main() {
   std::deque<double> values; // Создаем контейнер чисел с плавающей точкой
   std::ostream_iterator<double> output(std::cout, " ");

   // Вставляем элементы в values
   values.push_front(2.2);
   values.push_front(3.5);
   values.push_back(1.1);

   std::cout << "values contains: ";

   // Используем оператор индекса для доступа к элементам values
   for (size_t i{0}; i < values.size(); ++i) {
      std::cout << values[i] << ' ';
   }

   values.pop_front(); // Удаляем первый элемент
   std::cout << "\nAfter pop_front, values contains: ";
   std::ranges::copy(values, output);

   // Используем оператор индекса для изменения элемента в позиции 1
   values[1] = 5.4;
```

[1] «std::queue». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/container/queue.

[2] Это особенность конкретной реализации C++, но не требование стандарта.

```
30    std::cout << "\nAfter values[1] = 5.4, values contains: ";
31    std::ranges::copy(values, output);
32    std::cout << "\n";
33 }
```

```
values contains: 3.5 2.2 1.1
After pop_front, values contains: 2.2 1.1
After values[1] = 5.4, values contains: 2.2 5.4
```

Илл. 13.5 | Шаблон класса deque стандартной библиотеки

В строке 9 создается экземпляр контейнера `deque` для чисел с плавающей точкой, затем в строках 13–15 вызываются функции `push_front` и `push_back` для вставки элементов в начало и в конец контейнера.

В строках 20–22 все элементы выводятся на экран с помощью оператора индекса. В условии цикла используется функция `size`, чтобы предотвратить доступ к элементу за пределами контейнера `deque`. Здесь мы выбрали цикл `for` со счетчиком только для того, чтобы продемонстрировать оператор `[]`. Как правило, для обработки всех элементов контейнера предпочтительнее цикл `for` для диапазона значений.

В строке 24 мы используем функцию `pop_front`, демонстрируя удаление первого элемента контейнера. В строке 29 мы используем оператор индекса, чтобы получить значение *lvalue*, которое нужно для изменения элемента на позиции `1`.

13.9. Ассоциативные контейнеры

Ассоциативные контейнеры предоставляют прямой доступ для хранения и извлечения элементов с помощью ключей. Есть четыре упорядоченных ассоциативных контейнера: `multiset`, `set`, `multimap` и `map`. Каждый из них сохраняет свои ключи в отсортированном порядке. Есть также четыре соответствующих неупорядоченных ассоциативных контейнера: `unordered_multiset`, `unordered_set`, `unordered_multimap` и `unordered_map`, которые имеют большинство возможностей их упорядоченных аналогов. Основное различие между упорядоченными и неупорядоченными ассоциативными контейнерами заключается в том, что в упорядоченных контейнерах ключи расположены в отсортированном порядке. Если ваши ключи не являются сравнимыми, вам нужны неупорядоченные контейнеры. В этом разделе основное внимание будет уделено упорядоченным ассоциативным контейнерам.

Если нет необходимости в отсортированных ключах, неупорядоченные ассоциативные контейнеры обеспечивают высокое быстродействие поиска с помощью хеширования: $O(1)$ с возможным худшим значением $O(n)$; это значительно лучше, чем $O(\log n)$ при аналогичном поиске в упорядоченных ассоциативных контейнерах[1].

[1] «Containers Library». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/container.

Однако для этого требуется, чтобы ключи были хешируемыми. Требования к хешируемым типам вы найдете в документации к `std::hash`[1]. В разделе 13.13 вы познакомитесь с основами хеширования.

Итерации по упорядоченному ассоциативному контейнеру выполняются в порядке его сортировки. Классы `multiset` и `set` предназначены для операций с такими множествами значений, где сами значения являются ключами. Принципиальное различие между `multiset` и `set` заключается в том, что `multiset` допускает дублирование ключей, а в `set` дубликаты запрещены. Классы `multimap` и `map` предназначены для операций с такими множествами значений, где каждому значению соответствует ключ. Основное различие между `multimap` и `map` заключается в том, что `multimap` допускает дублирование ключей, а `map` может содержать только уникальные ключи. В дополнение к общим функциям контейнеров упорядоченные ассоциативные контейнеры поддерживают некоторые функции, специфичные для ассоциативных контейнеров. Вы можете объединить содержимое ассоциативных контейнеров одного и того же типа, используя функцию `merge` (C++17). Примеры упорядоченных ассоциативных контейнеров и их общих функций представлены в нескольких следующих разделах.

13.9.1. Ассоциативный контейнер multiset

Упорядоченный ассоциативный контейнер `multiset` (заголовок `<set>`) обеспечивает быстрое хранение и извлечение ключей с возможностью их дублирования. Порядок элементов определяет компаратор, который являстся функциональным объектом. **Функциональный объект (function object)** — это экземпляр класса, который имеет перегруженный оператор «круглые скобки», позволяющий «вызывать» объект как функцию. Например, контейнер `multiset`, содержащий целые числа, может быть упорядочен в порядке возрастания путем сортировки ключей с помощью компаратора (функционального объекта) `less<int>`, который знает, как сравнить два числа `int`, чтобы выяснить, является ли первое меньшим, чем второе. Это позволяет контейнеру `multiset` упорядочивать целочисленные элементы в порядке возрастания. В разделе 14.5 функциональные объекты рассматриваются подробнее. Здесь мы просто покажем использование `less<int>` при объявлении упорядоченных ассоциативных контейнеров.

Тип ключей во всех упорядоченных ассоциативных контейнерах должен поддерживать сравнение компаратором: ключи, сортируемые с помощью `less<T>`, должны поддерживать сравнение с помощью `operator<`. Если ключи, используемые в упорядоченных ассоциативных контейнерах, относятся к пользовательским типам данных, эти типы должны поддерживать операторы сравнения. Контейнер `multiset` поддерживает двунаправленные итераторы. Если порядок ключей не важен, лучше использовать контейнер `unordered_multiset` (заголовок `<unordered_set>`), но имейте в виду, что он поддерживает только однонаправленные итераторы.

[1] «std::hash». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/utility/hash.

Создание контейнера multiset

На илл. 13.6 показан упорядоченный ассоциативный контейнер `multiset` с целочисленными ключами, отсортированными в порядке возрастания. Контейнеры `multiset` и `set` (раздел 13.9.2) предоставляют одинаковую базовую функциональность. В строке 12 создается контейнер `multiset` с помощью функционального объекта `std::less<int>`, определяющего порядок сортировки ключей. Компилятор знает, что контейнер содержит значения типа `int`, поэтому вместо `std::less<int>` вы можете указать `std::less<>` — компилятор автоматически определит, что `std::less` сравнивает числа `int`. Кроме того, функциональный объект `std::less<>` используется по умолчанию для контейнера `multiset`, поэтому строку 12 можно упростить таким образом:

```
std::multiset<int> ints{}; // Контейнер multiset с элементами int
```

```
1 // fig13_06.cpp
2 // Демонстрация шаблона класса multiset стандартной библиотеки
3 #include <algorithm> // Алгоритм copy
4 #include <fmt/format.h> // Или в C++20: #include <format>
5 #include <iostream>
6 #include <iterator> // ostream_iterator
7 #include <ranges>
8 #include <set> // Определение шаблона класса multiset
9 #include <vector>
10
11 int main() {
12    std::multiset<int, std::less<int>> ints{}; // Контейнер с числами int
```

Илл. 13.6 | Демонстрация шаблона класса multiset стандартной библиотеки

Функция count контейнера multiset

В строке 13 вызывается функция `count` (есть во всех ассоциативных контейнерах) для подсчета количества вхождений числа `15` в контейнере.

```
13    std::cout << fmt::format("15s in ints: {}\n", ints.count(15));
14
```

```
15s in ints: 0
```

Функция insert контейнера multiset

В строках 16–17 одна из нескольких перегруженных версий функции `insert` дважды вставляет в контейнер значение `15`. Вторая версия функции `insert` принимает итератор и значение в качестве аргументов и начинает поиск точки вставки с указанной позиции итератора. Третья перегруженная версия функции `insert` использует два итератора, которые задают обычный диапазон для вставки значений из другого контейнера. О других перегрузках этой функции вы можете узнать здесь:

https://en.cppreference.com/w/cpp/container/multiset/insert

```
15    std::cout << "\nInserting two 15s into ints\n";
16    ints.insert(15); // Вставляем 15 в ints
17    ints.insert(15); // Вставляем 15 в ints
18    std::cout << fmt::format("15s in ints: {}\n\n", ints.count(15));
19
```

```
Inserting two 15s into ints
15s in ints: 2
```

Функция find контейнера multiset

В строках 21–28 вызывается функция `find` (строка 22), предоставляемая всеми ассоциативными контейнерами, для поиска значений `15` и `20`. Цикл `for` для диапазона значений перебирает элементы из списка `{15, 20}`. Функция `find` возвращает либо `iterator`, либо `const_iterator`, в зависимости от возможности изменения контейнера. Итератор указывает на позицию, в которой найдено значение. Если значение не найдено, функция `find` возвращает `iterator` или `const_iterator`, возвращенный функцией `end`, то есть указывающий на ограничитель контейнера. Обычно функция `find` используется, если требуется итератор, указывающий на найденный элемент. В этом примере мы могли бы использовать и функцию `count`, которая возвращает `0`, если, элемент не содержится в контейнере.

```
20    // Ищем числа 15 и 20 в ints; функция find возвращает итератор
21    for (int i : {15, 20}) {
22       if (auto result{ints.find(i)}; result != ints.end()) {
23          std::cout << fmt::format("Found {} in ints\n", i);
24       }
25       else {
26          std::cout << fmt::format("Did not find {} in ints\n", i);
27       }
28    }
29
```

```
Found 15 in ints
Did not find 20 in ints
```

Функция contains контейнера multiset (C++20)

В строках 31–38 мы вызываем введенную в C++20 функцию `contains` (строка 32), чтобы выяснить, есть ли в контейнере `multiset` числа `15` и `20`. Эта функция, предоставляемая всеми ассоциативными контейнерами, возвращает значение `bool`, указывающее, есть ли искомый элемент в контейнере. Цикл `for` для диапазона значений перебирает элементы из списка `{15, 20}`.

```
30    // Ищем числа 15 и 20 в ints; функция contains возвращает bool
31    for (int i : {15, 20}) {
32       if (ints.contains(i)) {
33          std::cout << fmt::format("Found {} in ints\n", i);
34       }
35       else {
```

```
36          std::cout << fmt::format("Did not find {} in ints\n", i);
37       }
38    }
39
```

```
Found 15 in ints
Did not find 20 in ints
```

Вставка элементов из другого контейнера в multiset

В строке 42 вызывается функция `insert` для вставки элементов вектора в контейнер, затем в строке 44 элементы контейнера копируются в стандартный поток вывода. Значения отображаются в порядке возрастания, поскольку `multiset` представляет собой упорядоченный контейнер, который по умолчанию хранит свои элементы в порядке возрастания. Обратите внимание на выражение в строке 44:

```
std::ostream_iterator<int>{std::cout, " "}
```

Оно создает временный `ostream_iterator` и немедленно передает его для копирования. Мы выбрали этот вариант, потому что `ostream_iterator` используется в этом примере только один раз.

```
40    // Вставляем элементы из вектора values в контейнер ints
41    const std::vector values{7, 22, 9, 1, 18, 30, 100, 22, 85, 13};
42    ints.insert(values.cbegin(), values.cend());
43    std::cout << "\nAfter insert, ints contains:\n";
44    std::ranges::copy(ints, std::ostream_iterator<int>{std::cout, " "});
45
```

```
After insert, ints contains:
1 7 9 13 15 15 18 22 22 30 85 100
```

Функции lower_bound и upper_bound контейнера multiset

В строке 49 вызываются функции `lower_bound` и `upper_bound` для поиска в контейнере `multiset` первого вхождения числа 22 и следующего элемента после последнего вхождения числа 22. Эти функции доступны всем упорядоченным ассоциативным контейнерам. Каждая из них возвращает `iterator` или `const_iterator`, указывающий на соответствующую позицию либо на ограничитель контейнера, если нужное значение отсутствует. Полученная в результате пара итераторов представляет границы обычного диапазона элементов, содержащего число 22.

```
46    // Находим в ints границы обычного диапазона, содержащего число 22
47    std::cout << fmt::format(
48       "\n\nlower_bound(22): {}\nupper_bound(22): {}\n\n",
49       *ints.lower_bound(22), *ints.upper_bound(22));
50
```

```
lower_bound(22): 22
upper_bound(22): 30
```

Объект pair и функция equal_range контейнера multiset

Функция `equal_range` возвращает объект `pair`, содержащий результаты вызова функций `lower_bound` и `upper_bound`. Объект `pair` объединяет два значения. В строке 52 создается и инициализируется объект `pair` с именем `p`. Мы используем ключевое слово `auto` для автоматического определения типа переменной по типу инициализатора. Объект `pair`, возвращаемый функцией `equal_range`, будет содержать два итератора: `iterator` или `const_iterator`, в зависимости от возможности изменения контейнера. Объект `pair` содержит два открытых элемента данных: `first` и `second`, их типы зависят от инициализаторов объекта.

```
51    // Снова находим в ints границы обычного диапазона, содержащего число 22
52    auto p{ints.equal_range(22)};
53    std::cout << fmt::format(
54       "lower_bound(22): {}\nupper_bound(22): {}\n",
55       *(p.first), *(p.second));
56 }
```

```
lower_bound(22): 22
upper_bound(22): 30
```

В строке 52 вызывается функция `equal_range` для определения границ обычного диапазона, содержащего число `22`. В строке 55 используются `p.first` и `p.second` для доступа к `lower_bound` и `upper_bound`. Мы разыменовали итераторы, чтобы вывести на экран значения из позиций, возвращенных функцией `equal_range`. В этом примере мы не делали проверку, но вы всегда должны проверять итераторы, возвращаемые функциями `lower_bound`, `upper_bound` и `equal_range`. Разыменовывать такой итератор можно только после проверки, подтвердившей, что он не указывает на ограничитель контейнера!

Гетерогенный поиск (C++14)

До C++14 при поиске ключа в ассоциативном контейнере аргумент, предоставляемый функции `find` или аналогичной функции поиска, должен был иметь тот же тип, что и ключ контейнера. Например, если у вас был ключ типа `string`, вы могли передать в функцию `find` строку в стиле C для ее поиска. В таких случаях аргумент конвертируется во временный объект, имеющий тип ключа (`string`), который передается в функцию `find`. В C++14 и выше аргумент функции `find` (и аналогичных функций) может быть любого типа, если есть перегруженные операторы сравнения, которые могут сравнивать значения типа аргумента со значениями типа ключа контейнера. Если эти операторы предоставлены, то временные объекты не создаются. Это называется **гетерогенным поиском (heterogeneous lookup)**.

13.9.2. Ассоциативный контейнер set

Ассоциативный контейнер `set` (заголовок `<set>`) используется для быстрого сохранения и извлечения уникальных ключей. Реализация `set` подобна реализации `multiset`, но с тем отличием, что контейнер `set` должен содержать уникальные клю-

чи. Когда в контейнер вставляется дубликат, он игнорируется. Это нормально для математического множества[1] и не является ошибкой. Контейнер `set` поддерживает двунаправленные итераторы. Если порядок ключей не важен, рассмотрите возможность использования контейнера `unordered_set` (заголовок `<unordered_set>`), но учитывайте, что этот контейнер поддерживает только однонаправленные итераторы, а его ключи должны быть хешируемыми.

Создание контейнера set

На илл. 13.7 показан контейнер `set`, содержащий числа с плавающей точкой. В строке 10 создается контейнер, тип элементов которого автоматически определяется с помощью CTAD. Строка 10 эквивалентна строке:

```
std::set<double, std::less<double>> doubles{2.1, 4.2, 9.5, 2.1, 3.7};
```

Конструктор объекта `initializer_list` вставляет все элементы в контейнер. В строке 14 используется алгоритм `copy` из пространства имен `std::ranges` для вывода содержимого контейнера. Обратите внимание, что значение `2.1` повторяется в списке инициализаторов, но включено в контейнер только один раз. Напомним, что в `set` не разрешены дубликаты.

```
 1 // fig13_07.cpp
 2 // Демонстрация шаблона класса set стандартной библиотеки
 3 #include <algorithm>
 4 #include <fmt/format.h> // Или в C++20: #include <format>
 5 #include <iostream>
 6 #include <iterator> // ostream_iterator
 7 #include <set>
 8
 9 int main() {
10    std::set doubles{2.1, 4.2, 9.5, 2.1, 3.7}; // Используем CTAD
11
12    std::ostream_iterator<double> output{std::cout, " "};
13    std::cout << "doubles contains: ";
14    std::ranges::copy(doubles, output);
15
```

```
doubles contains: 2.1 3.7 4.2 9.5
```

Илл. 13.7 | Демонстрация шаблона класса set стандартной библиотеки

Вставка нового значения в контейнер set

В строке 19 определяется и инициализируется объект `pair`, сохраняющий результат вызова функции `insert` контейнера `set`. Объект `pair` содержит итератор, указывающий на элемент в контейнере, и значение `bool`, указывающее на результат вставки: `true`, если значение элемента отсутствовало в контейнере, и `false`

[1] Английское слово «set» в данном случае означает «множество» в математическом смысле. — *Примеч. ред.*

в противном случае. Вызываемая в строке 19 функция `insert` вставляет значение `13.8` и возвращает объект `pair`, в котором `p.first` указывает на значение `13.8` в контейнере, а `p.second` имеет значение `true`, поскольку элемент был вставлен.

```
16    // Вставляем 13.8 в контейнер doubles; функция insert возвращает объект
17    // pair, в котором: p.first указывает на позицию значения 13.8 в doubles,
18    // p.second обозначает, было ли вставлено значение 13.8
19    auto p{doubles.insert(13.8)}; // Этого значения еще нет в контейнере
20    std::cout << fmt::format("\n{} {} inserted\n", *(p.first),
21       (p.second ? "was" : "was not"));
22    std::cout << "doubles contains: ";
23    std::ranges::copy(doubles, output);
24
```

```
13.8 was inserted
doubles contains: 2.1 3.7 4.2 9.5 13.8
```

Вставка дубликата в контейнер

В строке 26 осуществляется попытка вставить число `9.5`, которое уже есть в контейнере. Входные данные показывают, что число `9.5` не было вставлено, потому что контейнеры `set` не допускают дублирования ключей. В этом случае `p.first` в объекте `pair` указывает на существующий элемент контейнера со значением `9.5`, а `p.second` содержит `false`.

```
25    // Вставляем 9.5 в doubles
26    p = doubles.insert(9.5); // Это значение уже есть в контейнере
27    std::cout << fmt::format("\n{} {} inserted\n", *(p.first),
28       (p.second ? "was" : "was not"));
29    std::cout << "doubles contains: ";
30    std::ranges::copy(doubles, output);
31    std::cout << "\n";
32 }
```

```
9.5 was not inserted
doubles contains: 2.1 3.7 4.2 9.5 13.8
```

13.9.3. Ассоциативный контейнер multimap

Ассоциативный контейнер `multimap` используется для быстрого сохранения и извлечения ключей и связанных с ними значений; их называют **парами «ключ — значение» (key-value pairs)**. Многие функции, используемые с контейнерами `multiset` и `set`, также доступны для работы с контейнерами `multimap` и `map`. Каждый элемент контейнера `multimap` или `map` — это пара «ключ — значение». При вставке элементов в `multimap` или `map` вы используете объект `pair`, содержащий ключ и значение. Порядок ключей определяется компаратором. Например, в контейнере `multimap`, который использует в качестве ключей целые числа, они могут быть отсортированы в порядке возрастания с помощью функционального объекта `less<int>`.

Контейнер `multimap` поддерживает дубликаты ключей, поэтому с одним ключом может быть связано несколько значений. Это называется отношением «один ко многим». Например, в системе обработки транзакций платежных карт один счет может быть связан со многими транзакциями; в университете один студент может посещать много курсов, а один профессор может преподавать многим студентам. Контейнер `multimap` поддерживает двунаправленные итераторы.

Создание контейнера multimap, содержащего пары «ключ — значение»

На илл. 13.8 показан пример ассоциативного контейнера `multimap` (заголовок `<map>`). Если порядок ключей не важен, вы можете использовать `unordered_multimap` (заголовок `<unordered_map>`). Контейнер `multimap` реализован для эффективного поиска всех значений, связанных с заданным ключом. В строке 8 создается контейнер `multimap` с целыми числами в качестве ключей и соответствующими значениями с плавающей точкой, где элементы по умолчанию упорядочены в порядке возрастания. Эта строка эквивалентна следующей:

```
std::multimap<int, double, std::less<int>> pairs{};
```

```
1 // fig13_08.cpp
2 // Демонстрация шаблона класса multimap стандартной библиотеки
3 #include <fmt/format.h> // Или в C++20: #include <format>
4 #include <iostream>
5 #include <map> // Определение шаблона класса multimap
6
7 int main() {
8    std::multimap<int, double> pairs{}; // Создаем контейнер multimap
```

Илл. 13.8 | Демонстрация шаблона класса multimap стандартной библиотеки

Подсчет количества пар «ключ — значение» для заданного ключа

В строке 10 вызывается функция `count` для определения количества пар «ключ — значение» с ключом, равным `15`. Результат равен нулю, поскольку в данный момент контейнер пуст.

```
9    std::cout << fmt::format("Number of 15 keys in pairs: {}\n",
10   pairs.count(15));
11
```

```
Number of 15 keys in pairs: 0
```

Вставка пар «ключ — значение»

В строке 13 вызывается функция `insert` для вставки новой пары «ключ — значение» в контейнер `multimap`. Функция стандартной библиотеки `make_pair` создает объект `pair`, автоматически определяя типы его аргументов. В этом случае первый аргумент представляет ключ (`15`) типа `int`, а второй аргумент представляет значение (`99.3`) типа `double`. В строке 14 вставляется другой объект `pair` с ключом `15`

и значением 2.7. Затем в строках 15–16 выводится количество пар «ключ — значение», в которых ключ равен 15.

```
12    // Вставляем две пары "ключ - значение"
13    pairs.insert(std::make_pair(15, 99.3));
14    pairs.insert(std::make_pair(15, 2.7));
15    std::cout << fmt::format("Number of 15 keys in pairs: {}\n\n",
16       pairs.count(15));
17
```

```
Number of 15 keys in pairs: 2
```

Вставка пары «ключ — значение» скобочной инициализацией вместо make_pair

Вы можете использовать скобочную инициализацию для объектов pair, поэтому строки 13–14 можно записать в упрощенном виде:

```
pairs.insert({15, 99.3});
pairs.insert({15, 2.7});
```

В строках 19–23 в контейнер вставляются еще пять объектов pair. Инструкция for для диапазона значений в строках 28–30 выводит ключи и значения контейнера. Тип управляющей переменной цикла определяется автоматически: в данном случае это объект pair, содержащий целочисленный ключ и значение с плавающей точкой. В строке 29 осуществляется доступ к элементам каждого объекта pair. Обратите внимание, что ключи отображаются в порядке возрастания, потому что контейнер multimap обеспечивает сортировку ключей в порядке возрастания.

```
18 // Вставляем пять пар "ключ - значение"
19    pairs.insert({30, 111.11});
20    pairs.insert({10, 22.22});
21    pairs.insert({25, 33.333});
22    pairs.insert({20, 9.345});
23    pairs.insert({5, 77.54});
24
25    std::cout << "Multimap pairs contains:\nKey\tValue\n";
26
27    // Перебираем элементы контейнера (пары) в цикле
28    for (const auto& mapItem : pairs) {
29       std::cout << fmt::format("{}\t{}\n", mapItem.first, mapItem.second);
30    }
31 }
```

```
Multimap pairs contains:
Key     Value
5       77.54
10      22.22
15      99.3
15      2.7
```

```
20      9.345
25      33.333
30      111.11
```

Скобочная инициализация контейнера multimap (C++11)

В этом примере использовались отдельные вызовы функции `insert` для вставки пар «ключ — значение» в контейнер. Если эти пары известны заранее, можно использовать скобочную инициализацию при создании контейнера. Например, следующая инструкция инициализирует контейнер тремя парами «ключ — значение», помещенными во вложенные списки основного списка инициализаторов:

```
std::multimap<int, double> pairs{
   {10, 22.22}, {20, 9.345}, {5, 77.54}};
```

13.9.4. Ассоциативный контейнер map

Ассоциативный контейнер `map` (заголовок `<map>`) выполняет быстрое сохранение и извлечение уникальных ключей и связанных с ними значений. Дубликаты ключей не допускаются: с каждым ключом может быть связано только одно значение. Это называется отношением «один к одному». Например, компания, использующая уникальные номера сотрудников `100`, `200` и `300`, может хранить в контейнере `map` эти номера в парах с внутренними телефонами сотрудников: `4321`, `4115` и `5217` соответственно. Предоставление ключа в операторе индекса `[]` определяет позицию значения, связанного с этим ключом в контейнере. Если порядок ключей не важен, рассмотрите возможность использования контейнера `unordered_map` (заголовок `<unordered_map>`), но учитывайте, что ключи должны быть хешируемыми.

На илл. 13.9 показан пример контейнера `map` (строки 9–10). Только шесть из восьми первоначальных пар «ключ — значение» вставлены в контейнер, поскольку две пары из восьми имели дубликаты ключей. В отличие от похожих структур данных в некоторых других языках программирования, при вставке двух пар «ключ — значение» с одним и тем же ключом первое значение не заменяется вторым. Если вы вставляете пару «ключ — значение» с существующим в контейнере ключом, эта пара игнорируется.

```
 1 // fig13_09.cpp
 2 // Демонстрация шаблона класса map стандартной библиотеки
 3 #include <iostream>
 4 #include <fmt/format.h> // Или в C++20: #include <format>
 5 #include <map> // Определение шаблона класса map
 6
 7 int main() {
 8    // Создаем контейнер map; дубликаты ключей игнорируются
 9    std::map<int, double> pairs{{15, 2.7}, {30, 111.11}, {5, 1010.1},
10       {10, 22.22}, {25, 33.333}, {5, 77.54}, {20, 9.345}, {15, 99.3}};
11
```

Окончание ⇨

```
12    // Перебираем элементы контейнера (пары) в цикле
13    std::cout << "pairs contains:\nKey\tValue\n";
14    for (const auto& pair : pairs) {
15       std::cout << fmt::format("{}\t{}\n", pair.first, pair.second);
16    }
17
18    pairs[25] = 9999.99; // Изменяем значение с ключом 25
19    pairs[40] = 8765.43; // Вставляем значение с ключом 40
20
21    // Снова перебираем элементы контейнера (пары) в цикле
22    std::cout << "\nAfter updates, pairs contains:\nKey\tValue\n";
23    for (const auto& pair : pairs) {
24       std::cout << fmt::format("{}\t{}\n", pair.first, pair.second);
25    }
26 }
```

```
pairs contains:
Key     Value
5       1010.1
10      22.22
15      2.7
20      9.345
25      33.333
30      111.11

After updates, pairs contains:
Key     Value
5       1010.1
10      22.22
15      2.7
20      9.345
25      9999.99
30      111.11
40      8765.43
```

Илл. 13.9 | Демонстрация шаблона класса map стандартной библиотеки

В строках 18–19 используется оператор индекса `[]`. Когда индекс является ключом, существующим в контейнере, этот оператор возвращает ссылку на соответствующее значение. Когда индекс является ключом, отсутствующим в контейнере, оператор индекса добавляет новую пару «ключ — значение», состоящую из указанного ключа и значения по умолчанию для типа значений контейнера. В строке 18 значение с ключом `25` (ранее `33.333`, как указано в строке 10) заменяется на `9999.99`. В строке 19 в контейнер вставляется новая пара «ключ — значение».

13.10. Адаптеры контейнеров

В стандартной бибилиотеке есть три адаптера контейнеров: `stack`, `queue` и `priority_queue`. Адаптеры контейнеров не обеспечивают реализацию структур данных,

хранящих элементы, и не поддерживают итераторы. С помощью класса адаптера вы можете выбрать базовый последовательный контейнер или использовать контейнер по умолчанию: `deque` — для адаптеров `stack` и `queue`, `vector` — для адаптера `priority_queue`. Классы адаптеров предоставляют функции `push` и `pop`:

- `push` корректно вставляет элемент в соответствующий адаптеру контейнер;
- `pop` корректно удаляет элемент из соответствующего адаптеру контейнера.

Рассмотрим примеры работы с классами адаптеров.

13.10.1. Адаптер stack

Класс `stack` (заголовок `<stack>`) предоставляет функциональность стека, то есть позволяет добавлять и удалять элементы контейнера на одном конце (наверху). Стек — это структура данных, в которой первым обрабатывается последний поступивший элемент (last-in, first-out — LIFO). Стек может быть реализован с помощью контейнеров `vector`, `list` или `deque`. По умолчанию адаптер `stack` реализуется с помощью контейнера `deque`[1] и предоставляет следующие операции:

- `push` — для вставки элемента наверху стека; реализуется вызовом функции `push_back` контейнера;
- `emplace` — для создания элемента наверху стека;
- `pop` — для удаления элемента наверху стека; реализуется вызовом функции `pop_back` контейнера;
- `top` — для получения ссылки на элемент наверху стека; реализуется вызовом функции `back` контейнера;
- `empty` — для определения наличия элементов в стеке; реализуется вызовом функции `empty` контейнера;
- `size` — для получения количества элементов стека; реализуется вызовом функции `size` контейнера.

На илл. 13.10 показан пример работы со стеком. В этом примере создаются три объекта `stack`, содержащие числа `int` и использующие в качестве базовых структур данных контейнеры `deque` (строка 26), `vector` (строка 27) и `list` (строка 28).

```
1 // fig13_10.cpp
2 // Демонстрация класса адаптера stack стандартной библиотеки
3 #include <iostream>
4 #include <list> // Определение шаблона класса list
5 #include <ranges>
6 #include <stack> // Определение адаптера stack
7 #include <vector> // Определение шаблона класса vector
8
```

Окончание ⇨

[1] «std::stack». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/container/stack.

```
 9 // pushElements - обобщенная лямбда для добавления элементов в стек
10 auto pushElements = [](auto& stack) {
11     for (auto i : std::views::iota(0, 10)) {
12         stack.push(i); // Добавляем элемент в стек
13         std::cout << stack.top() << ' '; // Отображаем верхний элемент
14     }
15 };
16
17 // popElements - обобщенная лямбда для удаления элементов из стека
18 auto popElements = [](auto& stack) {
19     while (!stack.empty()) {
20         std::cout << stack.top() << ' '; // Отображаем верхний элемент
21         stack.pop(); // Удаляем верхний элемент
22     }
23 };
24
25 int main() {
26     std::stack<int> dequeStack{}; // Используем deque по умолчанию
27     std::stack<int, std::vector<int>> vectorStack{}; // Используем vector
28     std::stack<int, std::list<int>> listStack{}; // Используем list
29
30     // Добавляем числа от 0 до 9 в каждый стек
31     std::cout << "Pushing onto dequeStack: ";
32     pushElements(dequeStack);
33     std::cout << "\nPushing onto vectorStack: ";
34     pushElements(vectorStack);
35     std::cout << "\nPushing onto listStack: ";
36     pushElements(listStack);
37
38     // Отображаем элементы каждого стека, затем удаляем их
39     std::cout << "\n\nPopping from dequeStack: ";
40     popElements(dequeStack);
41     std::cout << "\nPopping from vectorStack: ";
42     popElements(vectorStack);
43     std::cout << "\nPopping from listStack: ";
44     popElements(listStack);
45     std::cout << "\n";
46 }
```

```
Pushing onto dequeStack: 0 1 2 3 4 5 6 7 8 9
Pushing onto vectorStack: 0 1 2 3 4 5 6 7 8 9
Pushing onto listStack: 0 1 2 3 4 5 6 7 8 9

Popping from dequeStack: 9 8 7 6 5 4 3 2 1 0
Popping from vectorStack: 9 8 7 6 5 4 3 2 1 0
Popping from listStack: 9 8 7 6 5 4 3 2 1 0
```

Илл. 13.10 | Демонстрация класса адаптера stack стандартной библиотеки

Обобщенное лямбда-выражение `pushElements` (строки 10–15) помещает в стек числа от `0` до `9`. В строках 32, 34 и 36 оно вызывается для каждого стека. В строке 12 вызывается функция `push` (доступная в каждом классе адаптера) для добавления

целого числа наверху стека. В строке 13 вызывается функция `top` для получения доступа к верхнему элементу стека и для его вывода на экран. Функция `top` не удаляет верхний элемент.

Обобщенное лямбда-выражение `popElements` (строки 18–23) извлекает элементы из стека. В строках 40, 42 и 44 оно вызывается для каждого стека. В строке 20 вызывается функция `top` для получения доступа к верхнему элементу стека и для его вывода на экран. В строке 21 вызывается функция `pop` (есть в каждом классе адаптера) для удаления верхнего элемента стека. Функция `pop` не возвращает значение, поэтому нужно вызвать функцию `top` для получения значения верхнего элемента перед его удалением.

13.10.2. Адаптер queue

Класс `queue` (заголовок `<queue>`) предоставляет функциональность очереди. Она похожа на обычную очередь ожидания. Первый элемент в очереди выходит из нее первым. Таким образом, `queue` — это структура данных «первым пришел — первым ушел» (FIFO). Класс `queue` позволяет вставлять элементы только в конец базовой структуры данных и извлекать элементы из начала. Адаптер `queue` хранит элементы в последовательных контейнерах `list` или `deque`, по умолчанию используется `deque`[1]. Адаптер `queue` предоставляет следующие операции:

- `push` — для вставки элемента в конец очереди; реализуется вызовом функции `push_back` контейнера;
- `emplace` — для создания элемента в конце очереди;
- `pop` — для удаления элемента в начале очереди; реализуется вызовом функции `pop_front` контейнера;
- `front` — для получения ссылки на первый элемент очереди; реализуется вызовом функции `front` контейнера;
- `back` — для получения ссылки на последний элемент в очереди; реализуется вызовом функции `back` контейнера;
- `empty` — для определения наличия элементов в очереди; вызывается функция `empty` контейнера;
- `size` — для получения количества элементов в очереди; вызывается функция `size` контейнера.

На илл. 13.11 показана работа с адаптером `queue`. В строке 7 создается экземпляр `queue`, содержащий числа с плавающей точкой. В строках 10–12 вызывается функция `push` для добавления элементов в очередь. В цикле `while` в строках 17–20 вызывается функция `empty` (есть во всех контейнерах) для определения наличия элементов (строка 17). Пока в очереди есть элементы, в строке 18 вызывается функция `front` для чтения и вывода (но не для удаления) первого элемента оче-

[1] «queue». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/container/queue.

реди. В строке 19 первый элемент удаляется функцией pop, которая есть во всех классах адаптеров.

```
1 // fig13_11.cpp
2 // Демонстрация класса адаптера queue стандартной библиотеки
3 #include <iostream>
4 #include <queue> // Определение адаптера queue
5
6 int main() {
7    std::queue<double> values{}; // Очередь из чисел с плавающей точкой
8
9    // Добавляем элементы в values
10   values.push(3.2);
11   values.push(9.8);
12   values.push(5.4);
13
14   std::cout << "Popping from values: ";
15
16   // Удаляем элементы из values
17   while (!values.empty()) {
18      std::cout << values.front() << ' '; // Выводим первый элемент
19      values.pop(); // Удаляем элемент
20   }
21
22   std::cout << "\n";
23 }
```

```
Popping from values: 3.2 9.8 5.4
```

Илл. 13.11 | Демонстрация класса адаптера queue стандартной библиотеки

13.10.3. Адаптер priority_queue

Класс `priority_queue` (заголовок `<queue>`) позволяет вставлять элементы в отсортированную базовую структуру данных и удалять элементы из начала базовой структуры данных. По умолчанию адаптер `priority_queue` хранит свои элементы в контейнере `vector`[1]. Элемент, добавленный в `priority_queue`, вставляется в порядке приоритета, поэтому элемент с высшим приоритетом будет удален первым. Обычно это достигается организацией элементов в виде **кучи (heap)**[2], всегда содержащей элемент с высшим приоритетом в начале структуры данных. По умолчанию сравнения элементов выполняются функциональным объектом: компаратором `less<T>`.

Адаптер `priority_queue` предоставляет следующие операции:

- `push` — для вставки элемента в соответствии с порядком приоритета;

[1] «std::priority_queue». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/container/priority_queue.

[2] Не следует путать с кучей динамически выделяемой памяти.

- `emplace` — для создания элемента без учета приоритета и для последующей пересортировки `priority_queue`;
- `pop` — для удаления элемента с высшим приоритетом;
- `top` — для получения ссылки на первый элемент `priority_queue`; реализуется вызовом функции `front` контейнера;
- `empty` — для определения наличия элементов в `priority_queue`; реализуется вызовом функции `empty` контейнера;
- `size` — для получения количества элементов в `priority_queue`; реализуется вызовом функции `size` контейнера.

На илл. 13.12 демонстрируется работа с адаптером `priority_queue`. В строке 7 создается экземпляр `priority_queue`, содержащий числа с плавающей точкой и использующий контейнер `vector` в качестве базовой структуры данных. В строках 10–12 вызывается функция `push` для добавления элементов в `priority_queue`. В строках 17–20 вызывается функция `empty` (есть во всех контейнерах) для определения наличия элементов в `priority_queue` (строка 17). При наличии элементов в строке 18 вызывается функция `top` для извлечения и вывода элемента с высшим приоритетом (то есть элемента с наибольшим значением). В строке 19 вызывается функция `pop`, которая есть во всех классах адаптеров, для удаления элемента с высшим приоритетом.

```
1 // fig13_12.cpp
2 // Демонстрация класса адаптера priority_queue стандартной библиотеки
3 #include <iostream>
4 #include <queue> // Определение адаптера priority_queue
5
6 int main() {
7    std::priority_queue<double> priorities; // Создаем priority_queue
8
9    // Добавляем элементы в priorities
10   priorities.push(3.2);
11   priorities.push(9.8);
12   priorities.push(5.4);
13
14   std::cout << "Popping from priorities: ";
15
16   // Удаляем элемент из priority_queue
17   while (!priorities.empty()) {
18      std::cout << priorities.top() << ' '; // Отображаем верхний элемент
19      priorities.pop(); // Удаляем верхний элемент
20   }
21
22   std::cout << "\n";
23 }
```

```
Popping from priorities: 9.8 5.4 3.2
```

Илл. 13.12 | Демонстрация класса адаптера priority_queue стандартной библиотеки

13.11. Псевдоконтейнер bitset

Класс `bitset` (заголовок `<bitset>`) позволяет легко создавать и обрабатывать **наборы битов (bit sets)**, которые часто применяются для удобного представления множества битовых флагов. Каждый объект `bitset` фиксируется по размеру во время компиляции. Класс `bitset` — это инструмент для операций с отдельными битами.

Объявление

```
bitset<size> b;
```

создает объект `bitset` с именем `b`, содержащий `size` битов, изначально «выключенных», то есть со значением `0` («off», `false`)

Инструкция

```
b.set(bitNumber);
```

«включает» бит `bitNumber`, то есть присваивает ему значение `1` («on», `true`). Вы также можете вызвать эту функцию со вторым аргументом типа `bool` для «включения» (`true`) или «выключения» (`false`) этого бита. Выражение `b.set()` «включает» все биты битового контейнера `b`.

Инструкция

```
b.reset(bitNumber);
```

«выключает» бит `bitNumber`, то есть присваивает ему значение `0` («off», `false`). Выражение `b.reset()` «выключает» все биты битового контейнера `b`.

Инструкция

```
b.flip(bitNumber);
```

переключает бит `bitNumber`: если бит «включен» (`1`), функция `flip` его «выключает» (`0`), и наоборот. Выражение `b.flip()` «переключает» все биты битового контейнера `b`, то есть присваивает им противоположные значения.

Выражение

```
b[bitNumber]
```

для битового контейнера `b`, если он не является константным, возвращает ссылку `std::bitset::reference`[1], позволяющую вам устанавливать значение `bool` для бита `bitNumber`; в противном случае возвращает копию текущего значения `bool` этого бита.

Инструкция

```
b.test(bitNumber);
```

сначала проверяет, находится ли аргумент в диапазоне битового контейнера. При условии, что `bitNumber` находится в диапазоне (на основе количества битов в `b`), функция `test` возвращает `true`, если этот бит «включен», или `false`, если он «вы-

[1] «std::bitset<N>::reference». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/utility/bitset/reference.

ключен». Если указано значение `bitNumber` вне диапазона, функция `test` выдает исключение `out_of_range`.

Выражение

```
b.size()
```

возвращает количество битов в битовом контейнере `b`.

Выражение

```
b.count()
```

возвращает количество «включенных» битов в битовом контейнере `b`.

Выражение

```
b.any()
```

(добавлено в C++11) возвращает `true`, если хотя бы один бит «включен».

Выражение

```
b.all()
```

(добавлено в C++11) возвращает `true`, если все биты «включены».

Выражение

```
b.none()
```

(добавлено в C++11) возвращает `true`, если ни один бит не «включен» (все биты имеют значение `false`).

Выражения

```
b == b1
b != b1
```

сравнивают два битовых контейнера на предмет равенства и неравенства соответственно.

Каждый из операторов побитового присваивания, `&=`, `|=` и `^=`, может использоваться для объединения битовых контейнеров. Например, выражение

```
b &= b1;
```

предоставляет побитовую операцию «И» между контейнерами `b` и `b1`, «включая» каждый бит битового контейнера `b`, если «включены» соответствующие биты в обоих контейнерах. Побитовая операция «ИЛИ»

```
b |= b1;
```

«включает» каждый бит битового контейнера `b`, если «включен» хотя бы один из соответствующих битов обоих битовых контейнеров. Побитовая операция «исключающее ИЛИ»

```
b ^= b2;
```

«включает» каждый бит битового контейнера `b`, если «включен» только один из соответствующих битов обоих битовых контейнеров (но не оба бита).

Вы также можете выполнить побитовую операцию «НЕ». Следующее выражение возвращает копию битового контейнера с противоположными значениями всех битов:

```
~b
```

Инструкция

```
b >>= n;
```

сдвигает биты вправо на n позиций.

Инструкция

```
b <<= n;
```

сдвигает биты влево на n позиций.

Выражения

```
b.to_string()
b.to_ulong()
b.to_ullong()
```

преобразуют битовый контейнер b в объекты `string`, `unsigned long` или `unsigned long long` соответственно.

13.12. Основы нотации «О-большое»

В этой главе вы встречались с обозначениями $O(1)$, $O(n)$ и $O(\log n)$. В этом разделе мы объясним их и представим еще два: $O(n^2)$ и $O(n \log n)$. Эти выражения, содержащие заглавную букву O, характеризуют вычислительную сложность алгоритма, то есть объем работы, выполняемый алгоритмом для обработки n элементов. Для современных компьютеров, выполняющих миллиарды операций в секунду, эти выражения помогают оценить длительность выполнения алгоритма в реальной программе: или почти мгновенно, или за несколько секунд, минут, часов, дней, месяцев, лет... или даже дольше. Конечно, вы предпочли бы алгоритмы, которые выполняются быстро даже с огромным количеством обрабатываемых элементов. Это особенно актуально для современных приложений, обрабатывающих большие данные. В реализациях контейнеров и алгоритмов стандартной библиотеки обозначения вычислительной сложности $O(1)$, $O(n)$, $O(\log n)$ и даже $O(n \log n)$, типичные для относительно хороших алгоритмов сортировки, считаются достаточно эффективными. Зато алгоритмы с вычислительной сложностью $O(n^2)$ или, хуже того, $O(2^n)$ и $O(n!)$ могут «задуматься» над обработкой разумного количества элементов на несколько столетий или тысячелетий. Скорее всего, вы не захотите писать такие алгоритмы.

Алгоритмы поиска имеют одну и ту же цель: найти в контейнере (для удобства обсуждения мы будем считать, что это контейнер `array`) элемент, соответствующий заданному ключу поиска, если такой элемент существует. Однако есть несколько особенностей, отличающих такие алгоритмы друг от друга. Основное различие заключается в объеме работы, необходимой для нахождения нужного элемента,

по сравнению с количеством элементов в контейнере. Один из способов описать этот объем работы — **нотация О-большое (Big O notation)**, показывающая вычислительную сложность алгоритма, решающего задачу. Для типичных алгоритмов поиска и сортировки время решения задачи зависит в основном от количества элементов.

Алгоритмы O(1)

Предположим, что алгоритм выполняет проверку равенства первого и второго элементов массива. Если в массиве 10 элементов, алгоритму потребуется только одно сравнение. Если в массиве 1000 элементов, алгоритму все равно потребуется только одно сравнение. Как видите, вычислительная сложность этого алгоритма не зависит от количества элементов. Говорят, что подобные алгоритмы имеют постоянное время выполнения, которое обозначается выражением $O(1)$. Оно произносится так: «вычислительная сложность первого порядка». Алгоритм с вычислительной сложностью $O(1)$ не обязательно должен требовать только одного сравнения. Обозначение $O(1)$ просто указывает, что количество операций является постоянным: оно не растет с увеличением размера массива. Алгоритм, осуществляющий проверку равенства первого элемента и любого из трех последующих элементов, тоже является алгоритмом $O(1)$, даже если выполняет три сравнения.

Алгоритмы O(n)

Алгоритм, проверяющий равенство первого элемента и любого другого элемента массива, требует не более $n - 1$ сравнений, где n — количество элементов массива. Если массив состоит из 10 элементов, алгоритму потребуется не более 9 сравнений. Если массив будет содержать 1000 элементов, алгоритму потребуется не более 999 сравнений. По мере увеличения количества элементов число n в выражении $n - 1$ «доминирует» и вычитание единицы становится несущественным. Нотация «О-большое» подразумевает выделение «доминирующих» элементов и игнорирование других элементов, роль которых уменьшается по мере роста n. По этой причине алгоритмы, требующие, как правило, $n - 1$ сравнений (например, описанные выше), обозначаются выражением $O(n)$. Оно произносится так: «вычислительная сложность порядка n». Алгоритмы $O(n)$ называются алгоритмами с линейным временем выполнения.

Алгоритмы O(n²)

Теперь представим себе алгоритм, проверяющий наличие дубликатов среди элементов массива. Первый элемент надо сравнить со всеми остальными элементами массива. Второй элемент надо сравнить со всеми элементами, кроме первого (это сравнение уже было произведено на первом этапе). Третий элемент надо сравнить со всеми элементами, кроме первых двух. В конечном итоге этот алгоритм выполнит $(n - 1) + (n - 2) + ... + 2 + 1$ или $n^2/2 - n/2$ сравнений. По мере увеличения количества элементов часть формулы, представляющая n^2, будет доминировать, а часть, представляющая n, окажется несущественной. И снова нотация «О-большое» выделяет доминирующую часть n^2, игнорируя $n/2$.

«О-большое» наглядно показывает, как растет время выполнения алгоритма по отношению к количеству обработанных элементов. Предположим, что алгоритм требует n^2 сравнений. Для 4 элементов алгоритму потребуется 16 сравнений, а для 8 элементов — 64 сравнения. В таком алгоритме увеличение количества элементов в 2 раза увеличивает количество сравнений в 4 раза. Рассмотрим аналогичный алгоритм, требующий $n^2/2$ сравнений. Для 4 элементов алгоритму потребуется 8 сравнений, а для 8 элементов — 32 сравнения. Опять же, увеличение количества элементов в 2 раза увеличивает количество сравнений в 4 раза. «Усилия» алгоритма растут аналогично квадрату числа n, поэтому в нотации «О-большое» игнорируется константа и вычислительная сложность обоих этих алгоритмов обозначается выражением $O(n^2)$. Это называется квадратичным временем выполнения и произносится так: «вычислительная сложность порядка n в квадрате».

Эффективность линейного поиска O(n)

Алгоритмы линейного поиска, обычно использующиеся для поиска элемента в неотсортированном массиве, выполняются за время $O(n)$. Наихудшим сценарием в этом алгоритме является проверка каждого элемента массива для определения наличия ключа поиска в массиве. Если размер массива удваивается, количество сравнений, выполняемых алгоритмом, тоже удваивается. Линейный поиск может обеспечить высокое быстродействие, если искомый элемент находится в начале или рядом с началом массива. Но нам требуются алгоритмы, достаточно хорошо выполняющие любой поисковый запрос, даже если соответствующий ключу элемент находится в конце массива.

Алгоритмы линейного поиска легко создавать, но они могут уступать другими алгоритмам, особенно когда n становится большим. Если программа должна выполнять много поисковых запросов в больших массивах, лучше реализовать более эффективный алгоритм бинарного поиска (`binary_search`), который мы будем использовать в главе 14.

Иногда простые алгоритмы работают плохо. Их достоинство часто заключается в простоте создания, тестирования и отладки. Иногда для достижения максимального быстродействия необходимо создание более сложных алгоритмов.

Эффективность бинарного поиска O(log n)

В худшем случае поиск в отсортированном массиве из 1023 элементов (то есть $2^{10} - 1$) займет всего 10 сравнений при использовании бинарного поиска. Алгоритм сравнивает искомый элемент со средним элементом отсортированного массива. Если элементы равны, поиск окончен. Скорее всего, искомый элемент будет больше или меньше среднего элемента. Если он больше, мы можем исключить первую половину массива из процесса поиска. В противном случае мы можем исключить вторую половину массива. Это создает эффект сокращения вдвое, поэтому на следующих этапах нам нужно выполнить поиск среди 511, 255, 127, 63, 31, 15, 7, 3 и 1 элементов. Число 1023 (то есть $2^{10} - 1$) нужно уменьшить вдвое всего 10 раз, чтобы либо найти искомый элемент, либо убедиться в его отсутствии

в массиве. Деление на 2 эквивалентно одному сравнению в алгоритмах бинарного поиска. Таким образом, в массиве из 1 048 575 (то есть $2^{20} - 1$) элементов нужно не более 20 сравнений для поиска ключа, а в массиве из миллиарда элементов потребуется максимум 30 сравнений. Это огромный прирост быстродействия по сравнению с линейным поиском. Для массива из миллиарда элементов это будет разница между средним значением в 500 000 000 сравнений для линейного поиска и максимальным значением в 30 сравнений для бинарного поиска! Максимальное количество сравнений, необходимых для двоичного поиска элемента в любом отсортированном массиве, равно показателю наименьшей степени числа 2, превышающей количество элементов в массиве, что обозначается как $\log_2 n$. Все логарифмы увеличиваются «примерно с одинаковой скоростью», поэтому в терминах нотации «О-большое» основание логарифма можно опустить и описать такие алгоритмы выражением $O(\log n)$. Это называется логарифмическим временем выполнения и произносится так: «вычислительная сложность порядка логарифма n».

В следующей таблице мы перечислили самые распространенные обозначения нотации «О-большое» и несколько значений n, чтобы подчеркнуть различия в темпах роста. Интерпретируя каждое значение в таблице как время, затраченное на вычисления, вы можете понять, почему надо избегать алгоритмов $O(n^2)$.

n =	O(1)	O(log n)	O(n)	O(n log n)	$O(n^2)$
1	1	0	1	0	1
2	1	1	2	2	4
3	1	1	3	3	9
4	1	1	4	4	16
5	1	1	5	5	25
10	1	1	10	10	100
100	1	2	100	200	10 000
1000	1	3	1000	3000	10^6
1 000 000	1	6	1 000 000	6 000 000	10^{12}

13.13. Основы хеш-таблиц

При создании объектов вашей программе может потребоваться быстрый доступ к элементам для записи и чтения данных. Обработка данных, находящихся в массивах, будет эффективной, если один из элементов данных совпадает с индексом, а все индексы уникальны и хорошо упакованы. Предположим, у вас есть 100 сотрудников, имеющих номера социального страхования, и вы хотите обрабатывать данные о сотрудниках, используя эти номера в качестве индексов массива. Учитывая непрерывность последовательности индексов, вам понадобится массив, содержащий почти 900 миллионов элементов, поскольку номера социального

страхования являются 9-значными и начинаются с последовательностей цифр от 001 до 899 (кроме 666):

https://www.ssa.gov/employer/randomization.html

Это непрактично для большинства приложений, использующих номера социального страхования в качестве ключей. Только программа, действительно обрабатывающая гигантский массив, могла бы достичь высокого быстродействия как при хранении, так и при поиске записей о сотрудниках, просто используя номер социального страхования в качестве индекса массива.

Программисты часто сталкиваются с подобными проблемами. Ключи могут представлять неподходящий тип данных (например, отрицательные целые числа, которые не могут быть индексами массива), или ключи подходящего типа могут быть разбросаны в слишком большом диапазоне (например, номера социального страхования сотрудников маленькой компании). Необходима высокоскоростная схема преобразования ключей, таких как номера социального страхования, инвентарные номера и т. п., в уникальные индексы массива, имеющего приемлемый размер. И когда программе необходимо сохранить элемент данных, эта схема преобразует ключ в индекс массива (этот процесс называется хешированием), после чего элемент может быть сохранен в массиве. Извлечение данных выполняется таким же образом: когда у программы есть ключ, для которого требуется извлечь данные, она просто преобразует (хеширует) ключ, получая индекс элемента массива для обработки данных.

Почему это называется **хешированием (hashing)**[1]? Когда мы преобразуем ключ в индекс массива, мы выполняем заданный алгоритм, «перемещивающий» биты исходного числа (ключа) таким образом, чтобы получилось соответствующее ему хешированное число. Оно не имеет никакого реального смысла, это просто удобная «метка», по которой можно быстро найти нужный элемент данных.

Сбой в этой схеме называется **коллизией (collision)** — когда два разных ключа хешируются в одну и ту же ячейку (в один и тот же элемент) массива. Мы не можем хранить два значения в одной ячейке, поэтому нам понадобятся альтернативные индексы для всех значений, кроме первого, которое хешируется в данный индекс массива. Для этого существует множество схем. Одна из них — «повторное хеширование» — выполнение повторного хеширования на основе результатов предыдущего, чтобы найти в массиве свободную ячейку. Хеширование предназначено для распределения значений по всей таблице, поэтому свободная ячейка в итоге будет найдена: после одной или нескольких повторных операций хеширования.

В другой схеме, устраняющей коллизию, используется только одна операция хеширования для получения свободной ячейки. Если эта ячейка уже содержит данные, выполняется последовательный поиск по следующим ячейкам, пока не будет найдено свободное место. Извлечение данных работает таким же образом: ключ хешируется один раз для определения начальной позиции, затем проверяется наличие в ней нужных данных. Если данные верны, поиск будет завершен; в про-

[1] «Hashing» (англ.) — «перемешивание», «измельчение». — *Примеч. ред.*

тивном случае выполняется последовательный поиск в следующих ячейках, пока не будут найдены нужные данные. Популярным решением коллизий хеш-таблиц является создание **хеш-корзины (hash bucket)** для каждой ячейки таблицы — это, как правило, связный список всех пар «ключ — значение», хешируемых в определенную ячейку.

Коэффициент заполнения хеш-таблицы влияет на быстродействие схем хеширования. Коэффициент заполнения — это отношение количества занятых ячеек к общему количеству ячеек в хеш-таблице. Чем ближе это соотношение к 1.0, тем больше вероятность коллизии, замедляющей операции с данными.

Коэффициент заполнения хеш-таблицы является классическим примером компромисса между потребляемым объемом памяти и быстродействием. Увеличивая коэффициент заполнения, мы эффективнее используем память, но программа работает медленнее из-за увеличения коллизий хеширования. Уменьшая коэффициент заполнения, мы уменьшаем вероятность коллизий хеширования, что повышает скорость работы программы, но нерационально используем память, потому что большая часть хеш-таблицы остается пустой.

Ассоциативные контейнеры `unordered_set`, `unordered_multiset`, `unordered_map` и `unordered_multimap` реализованы «под капотом» компилятора в виде хеш-таблиц. Когда вы используете эти контейнеры, вы получаете преимущество высокоскоростной обработки данных без необходимости создавать ваши собственные механизмы хеширования. Это классический пример повторного использования кода.

Еще одно применение хеширования связано с виртуальной памятью[1]. Компания VMware[2] создает системы виртуализации. Например, на наших компьютерах с macOS мы используем VMware Fusion, чтобы запускать Windows и Linux параллельно с macOS. Все три операционные системы поддерживают виртуальную память. Программы и данные хранятся на большом дисковом накопителе и копируются для выполнения в ограниченную по объему оперативную память (а затем в более быструю и «тесную» кэш-память). Эффективность этой технологии основана на том, что в оперативной памяти (и в кэш-памяти) должны находиться только те фрагменты программы, которые выполняются в данный период времени. Чтобы находить нужные фрагменты в виртуальной памяти, необходим поиск в очень больших структурах данных, на которых основана виртуальная память. VMware изучает методы хэширования, оптимизированные для поддержки буферов ассоциативной трансляции (специальных структур кэш-памяти в процессорах) с целью ускорения поиска необходимых фрагментов в виртуальной памяти[3].

---

[1] «Virtual Memory». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Virtual_memory.

[2] «VMware: About Us». Был доступен: 15 сентября 2023 г. https://www.vmware.com/company.html.

[3] Алекс Конвей (Alex Conway), Роб Джонсон (Rob Johnson), Джейнел Ганди (Jayneel Gandhi) и др., «Hash-Based Virtual Memory: Reducing the Costs of Virtual Memory Through Better Data Structures». VMware. Был доступен: 15 сентября 2023 г. https://research.vmware.com/projects/hashed-p-o-t-a-t-o-e.

13.14. Итоги

В этой главе мы рассмотрели три ключевых компонента стандартной библиотеки: контейнеры, итераторы и алгоритмы. Вы узнали о линейных последовательных контейнерах `array` (представленных в главе 6), `vector`, `deque`, `forward_list` и `list`, представляющих линейные структуры данных. Мы обсудили нелинейные ассоциативные контейнеры `set`, `multiset`, `map`, `multimap` и их неупорядоченные версии. Вы узнали, что адаптеры контейнеров `stack`, `queue` и `priority_queue` могут использоваться для ограничения операций с последовательными контейнерами `vector`, `deque` и `list` при реализации специализированных структур данных, предоставляемых адаптерами контейнеров. Вы изучили все категории итераторов и узнали, что каждый алгоритм стандартной библиотеки может использоваться с любым контейнером, который поддерживает минимальную функциональность итератора, требуемую алгоритмом. Мы провели различие между обычными диапазонами и диапазонами C++20, а затем продемонстрировали алгоритм `std::ranges::copy`. Вы узнали о функциях класса `bitset`, который позволяет легко создавать наборы битов и работать с отдельными битами примерно так же, как с элементами контейнеров.

В главе 14 мы продолжим обсуждение контейнеров, итераторов и алгоритмов стандартной библиотеки, причем рассмотрим алгоритмы значительно подробнее. Вы увидите, что в стандартной библиотеке C++20 широко используются концепты. Мы сформулируем минимальные требования к итераторам, определяющие, какие контейнеры совместимы с каждым из алгоритмов. Мы продолжим тему лямбда-выражений и покажем, что алгоритмам можно передавать указатели на функции и функциональные объекты — экземпляры классов, перегружающих оператор вызова функции (оператор «скобки»). А в главе 15 вы увидите, как в игру вступают концепты, когда вам надо создавать пользовательские контейнеры и итераторы.

14

Алгоритмы стандартной библиотеки, диапазоны и представления C++20

В этой главе вы:

- узнаете о минимальных требованиях к итераторам для работы с контейнерами и алгоритмами стандартной библиотеки;
- научитесь создавать лямбда-выражения, захватывающие локальные переменные для их использования в теле лямбда-выражений;
- научитесь использовать алгоритмы C++20 из пространства имен `std::ranges`;
- познакомитесь с концептами C++20, определяющими минимальные требования к итераторам алгоритмов библиотеки `std::ranges`;
- сравните новые алгоритмы C++20 из пространства имен `std::ranges` со старыми алгоритмами обычных диапазонов из пространства имен `std`;
- научитесь использовать итераторы с алгоритмами для обработки элементов контейнеров стандартной библиотеки;
- научитесь передавать лямбда-выражения, указатели на функции и функциональные объекты алгоритмам стандартной библиотеки;
- научитесь использовать проекции для преобразования объектов в диапазоне при их обработке алгоритмами библиотеки `std::ranges`;
- научитесь использовать представления C++20 и ленивые вычисления с диапазонами C++20;
- узнаете о новых возможностях диапазонов, которые ожидаются в C++23;
- познакомитесь с параллельными алгоритмами, повышающими быстродействие (затем мы рассмотрим их подробнее в главе 17).

14.1. Введение

Эта глава является справочным материалом по многим алгоритмам стандартной библиотеки C++. Мы сосредоточимся на таких операциях с контейнерами, как заполнение значениями, генерация значений, сравнение отдельных элементов или целых контейнеров, удаление элементов, замена элементов, математические операции, поиск, сортировка, обмен значениями между элементами, копирование, слияние, операции с множествами, вычисление минимальных и максимальных значений, определение границ и т. д. Стандартная библиотека предоставляет множество готовых шаблонных алгоритмов:

- 90 алгоритмов в заголовке `<algorithm>` пространства имен `std`; 82 из них также перегружены в пространстве имен C++20 `std::ranges`;
- 11 алгоритмов в заголовке `<numeric>` пространства имен `std`;
- 14 алгоритмов в заголовке `<memory>` пространства имен `std`; все они также перегружены в пространстве имен C++20 `std::ranges`.

Многие алгоритмы были добавлены в C++11 и C++20, а некоторые — в C++17. Здесь приведен полный список алгоритмов с их описаниями:

https://en.cppreference.com/w/cpp/algorithm

Минимальные требования к алгоритмам и концептам C++20

Каждый алгоритм стандартной библиотеки имеет минимальные требования, по которым вы можете определить, какие контейнеры, итераторы и функции вы можете передавать этому алгоритму. Алгоритмы диапазонов C++20 из пространства имен `std::ranges` определяют эти требования с помощью **концептов (concepts)** C++20. Мы будем кратко рассматривать концепты по мере необходимости, чтобы вы понимали требования к работе с этими алгоритмами. Более подробно мы рассмотрим концепты в главе 15 при создании шаблонов.

Алгоритмы диапазонов C++20 в сравнении с прежними алгоритмами обычных диапазонов

Большинство алгоритмов, представленных в этой главе, перегружены:

- в пространстве имен `std::ranges` для использования их с диапазонами C++20;
- в пространстве имен `std` для использования их с обычными диапазонами — парами итераторов, представляющих первый элемент обычного диапазона и следующий элемент после последнего элемента диапазона.

В этой главе мы сосредоточимся на версиях из библиотеки `ranges` C++20[1, 2].

---

[1] Тристан Бриндл (Tristan Brindle), «An Overview of Standard Ranges», 29 сентября 2019 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=SYLgG7Q5Zws.

[2] Тристан Бриндл, «C++20 Ranges in Practice», 8 октября 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=d_E-VLyUnzc.

Лямбды, указатели на функции и функциональные объекты

В разделе 6.14.2 мы представили вам лямбда-выражения. В разделе 14.3 мы вернемся к ним и детальнее рассмотрим их возможности. Как вы увидите, различные алгоритмы могут получать в качестве аргумента лямбду, или указатель на функцию, или функциональный объект. Большинство примеров в этой главе содержат лямбды, потому что они удобны для решения локальных задач. В разделе 14.5 вы увидите, что лямбда-выражение можно заменить указателем на функцию или функциональным объектом (функтором). Класс функционального объекта перегружает функцию `operator()`, позволяя использовать имя объекта в качестве имени функции, например: *имяОбъекта*(*аргументы*). На протяжении всей этой главы, когда мы говорим о возможности алгоритма получать функцию в качестве аргумента, мы имеем в виду, что он может получать как функцию, так и лямбда-выражение или функциональный объект.

Представления C++20 и программирование в функциональном стиле с ленивыми вычислениями

Как и многие современные языки, C++ дает вам возможность программировать в функциональном стиле, с которым вы познакомились в разделе 6.14.3. В частности, мы продемонстрировали операции фильтрации, преобразования и свертки. В разделе 14.7 мы расскажем о программировании в функциональном стиле с помощью новой библиотеки `ranges` (C++20).

Параллельные алгоритмы

В C++17 были введены перегруженные параллельные (то есть оптимизированные для многоядерных процессоров) версии 69 алгоритмов стандартной библиотеки из заголовка `<algorithm>`, значительно повышающие быстродействие программ. Эти версии кратко рассматриваются в разделе 14.8. Мы перечисляем алгоритмы с параллельными версиями в сводных таблицах алгоритмов стандартной библиотеки в разделе 14.9. В главе 17 будет продемонстрирована работа нескольких параллельных алгоритмов. В этой главе мы будем использовать функции из заголовка `<chrono>`, чтобы измерять время, потраченное алгоритмом стандартной библиотеки, выполняемым сначала последовательно (на одном ядре), а потом параллельно (на нескольких ядрах). Вы увидите существенный прирост быстродействия.

Нововведения в диапазонах C++23

Некоторые алгоритмы C++20, включая алгоритмы из заголовка `<numeric>` и параллельные алгоритмы из заголовка `<algorithm>`, не имеют перегруженных версий в пространстве имен `std::ranges`. В разделе 14.10 рассматриваются изменения, ожидаемые в C++23, и упоминается проект с открытым исходным кодом `rangesnext`[1], содержащий реализации некоторых предлагаемых обновлений.

---

[1] Корентен Жабо (Corentin Jabot), «Ranges for C++23». Был доступен: 15 сентября 2023 г. https://github.com/cor3ntin/rangesnext.

14.2. Требования алгоритмов: концепты (C++20)

В стандартной библиотеке C++ контейнеры отделены от алгоритмов, обрабатывающих эти контейнеры. Большинство алгоритмов работают с элементами контейнеров косвенно, через итераторы. Такая архитектура упрощает написание универсальных алгоритмов для различных контейнеров. Это сильная сторона алгоритмов стандартной библиотеки.

Шаблоны классов контейнеров и соответствующие им шаблоны классов итераторов обычно находятся в одном и том же заголовке. Например, заголовок `<vector>` содержит шаблоны и для класса `vector`, и для класса его итераторов. Контейнер внутренне создает объекты своего класса итераторов и возвращает их с помощью функций контейнера, таких как `begin`, `end`, `cbegin` и `cend`.

Требования к итераторам

Каждый алгоритм может работать с любым контейнером, соответствующим минимальным требованиям этого алгоритма к итераторам[1]. Это сделано для максимизации повторного использования кода. Например, алгоритм, которому нужны однонаправленные итераторы, может работать с любым контейнером, предоставляющим как минимум эти итераторы. Все алгоритмы стандартной библиотеки могут работать с векторами (`vector`) и массивами (`array`), поскольку эти контейнеры поддерживают непрерывные итераторы, обеспечивающие все итерационные операции, о которых мы рассказали в разделе 13.3.

До C++20:

- в документации каждого контейнера был указан поддерживаемый уровень итератора;
- в документации каждого алгоритма были указаны минимальные требования к итератору.

Вы должны придерживаться документированных требований алгоритма, используя только контейнеры и итераторы, соответствующие этим требованиям. Если вы передадите алгоритму неправильный тип итератора, компилятор подставит этот тип в определение шаблона алгоритма и, как заметил Бьерн Страуструп, выдаст «поразительно плохие сообщения об ошибках»[2].

Концепты C++20

Одна из «большой четверки» новинок C++20 — это **концепты (concepts)**. Так называется технология ограничения типов, используемых с шаблонами. Страуструп указывает, что «концепты дополняют шаблоны C++, как первоначально

---

[1] Александр Степанов и Мэн Ли (Meng Lee), «The Standard Template Library, Section 2: Structure of the Library», 31 октября 1995 г. Был доступен: 15 сентября 2023 г. http://stepanovpapers.com/STL/DOC.PDF.

[2] Бьерн Страуструп (Bjarne Stroustrup), «Concepts: The Future of Generic Programming — 1. A Bit of Background», 31 января 2017 г. Был доступен: 15 сентября 2023 г. http://wg21.link/p0557r0.

предполагалось»[1] десятилетия назад. Каждый концепт определяет требования к типу или взаимосвязь между типами[2].

Когда параметр алгоритма ограничен концептом, компилятор сначала проверяет тип переданного вами аргумента на соответствие требованиям этого концепта и только потом подставляет этот аргумент в шаблон. Концепты дают вам большое преимущество: если тип аргумента не подошел к требованиям концепта, компилятор выдаст меньше сообщений об ошибках и эти сообщения будут более понятными, чем при работе со старыми алгоритмами для обычных диапазонов. Это упрощает выявление ошибок для последующего исправления кода.

Основное внимание в этой главе уделяется новым алгоритмам диапазонов C++20. Эти алгоритмы ограничены многими предопределенными концептами C++20. В стандарте существует 76 предопределенных концептов[3,4]. В этой главе мы будем использовать шаблоны стандартной библиотеки, как уже делали в главах 6 и 13. Но мы пока не будем использовать концепты в примерах кода. Программисты, создавшие алгоритмы диапазонов C++20, уже включили концепты в сигнатуры шаблонов функций алгоритмов, установив таким образом точные требования к итераторам и диапазонам, с которыми работают эти алгоритмы.

Вызывая любой алгоритм библиотеки `ranges` C++20, вы должны передать ему аргументы, соответствующие его требованиям к контейнерам и итераторам. Для всех алгоритмов, рассматриваемых в этой главе, мы назовем предопределенные имена концептов, указанные в прототипах этих алгоритмов, и кратко объясним, как они ограничивают аргументы этих алгоритмов. А в главе 15 мы рассмотрим шаблоны с точки зрения разработчика, демонстрируя практическую реализацию пользовательских шаблонов с концептами.

Концепты итераторов C++20

Просматривая документацию на алгоритмы диапазонов C++20, вы часто будете видеть в прототипах следующие концепты итераторов C++20, которые определены в пространстве имен `std` в заголовке `<iterator>`:

- `input_iterator` — итератор ввода;
- `output_iterator` — итератор вывода;
- `forward_iterator` — однонаправленный итератор;

---

[1] Бьерн Страуструп, «Concepts: The Future of Generic Programming — Conclusion», 31 января 2017 г. Был доступен: 15 сентября 2023 г. http://wg21.link/p0557r0.

[2] Бьерн Страуструп, «Concepts: The Future of Generic Programming — 3.1 Specifying Template Interfaces», 31 января 2017 г. Был доступен: 15 сентября 2023 г. http://wg21.link/p0557r0.

[3] «Index of Library Concepts». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/conceptindex.

[4] Стандарт также определяет более 30 концептов «только для демонстрационных целей», которые находятся в процессе обсуждения. Ответ от 28 декабря 2015 г. на вопрос «Exposition-Only in the C++ Standard?». Был доступен: 15 сентября 2023 г. https://stackoverflow.com/questions/34493104/exposition-only-in-the-c-standard.

- `bidirectional_iterator` — двунаправленный итератор;
- `random_access_iterator` — итератор произвольного доступа;
- `contiguous_iterator` — непрерывный итератор.

Они определяют требования к типам категорий итераторов, представленных в главе 13. Полный список концептов итераторов см. в разделе «C++20 iterator concepts» по адресу:

https://en.cppreference.com/w/cpp/iterator

Концепты диапазонов C++20

Концепт диапазона описывает тип с **начальным итератором (begin iterator)** и **конечным ограничителем (end sentinel)**, которые могут быть разных типов. Вы часто будете видеть следующие концепты диапазонов C++20 в прототипах алгоритмов диапазонов C++20:

- `input_range` — диапазон, поддерживающий итераторы ввода (`input_iterator`);
- `output_range` — диапазон, поддерживающий итераторы вывода (`output_iterator`);
- `forward_range` — диапазон, поддерживающий однонаправленные итераторы (`forward_iterator`);
- `bidirectional_range` — диапазон, поддерживающий двунаправленные итераторы (`bidirectional_iterator`);
- `random_access_range` — диапазон, поддерживающий итераторы произвольного доступа (`random_access_iterator`);
- `contiguous_range` — диапазон, поддерживающий непрерывные итераторы (`contiguous_iterator`).

Они определены в пространстве имен `std::ranges` в заголовке `<ranges>` и устанавливают требования к диапазонам, поддерживающим категории итераторов, рассмотренные в главе 13. В следующих главах мы обсудим и некоторые другие концепты.

14.3. Лямбда-выражения и алгоритмы

Вы можете модифицировать работу многих алгоритмов стандартной библиотеки, передавая функцию в качестве аргумента. Вы видели в разделе 6.14.2, что лямбда-выражения являются анонимными функциями — как правило, внутри других функций[1]. В разделе 8.19 вы видели, что лямбды могут обрабатывать локальные переменные внешней функции. Обычно мы передаем алгоритмам стандартной библиотеки лямбды, потому что они удобны для небольших задач.

---

[1] Лямбды также могут быть определены в классах или пространствах имен. Подробнее: https://en.cppreference.com/w/cpp/language/lambda.

На илл. 14.1 представлены версии алгоритмов `copy` и `for_each` стандартной библиотеки из пространства имен `std::ranges` (C++20). Напомним, что алгоритм `for_each` получает в качестве одного из своих аргументов функцию, определяющую задачу, выполняемую над каждым элементом контейнера.

```
1 // fig14_01.cpp
2 // Лямбда-выражения
3 #include <algorithm>
4 #include <array>
5 #include <iostream>
6 #include <iterator>
7
8 int main() {
9    std::array values{1, 2, 3, 4}; // Инициализация массива values
10   std::ostream_iterator<int> output{std::cout, " "};
11
12   std::cout << "values contains: ";
13   std::ranges::copy(values, output);
14
```

```
values contains: 1 2 3 4
```

Илл. 14.1 | Лямбда-выражения

Алгоритм copy и требования к итераторам обычных диапазонов в сравнении с требованиями к итераторам диапазонов C++20

В строке 9 создается массив, содержащий целые числа. В строке 13 он выводится на экран с помощью `ostream_iterator` и алгоритма `copy`[1] из пространства имен `std::ranges` (C++20). Напомним, что в разделе 13.6.2 алгоритм `copy` для обычных диапазонов из пространства имен `std` вызывался таким образом:

```
std::copy(integers.cbegin(), integers.cend(), output);
```

В документации этого алгоритма указано, что первые два его аргумента должны быть итераторами ввода, представляющими начало и конец копируемого обычного диапазона. Третьим аргументом должен быть итератор вывода, указывающий, куда надо скопировать элементы.

Теперь мы вызываем в строке 13 алгоритм `copy` из пространства имен `std::ranges` всего с двумя аргументами: контейнером `values` и `ostream_iterator`. Эта версия алгоритма самостоятельно определяет начало и конец контейнера `values`, вызывая:

```
std::ranges::begin(values)
```

а также:

```
std::ranges::end(values)
```

[1] «std::ranges::copy, std::ranges::copy_if, std::ranges::copy_result, std::ranges::copy_if_result». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/algorithm/ranges/copy.

Любой контейнер, поддерживающий итераторы `begin` и `end`, можно рассматривать как диапазон C++20. Первым аргументом алгоритма `std::ranges::copy` должен быть диапазон `input_range` (то есть диапазон, соответствующий этому концепту), а вторым аргументом — итератор вывода. Здесь мы выполняем копирование в стандартный поток вывода, но мы также можем копировать в другой диапазон. К итератору вывода предъявляется несколько требований:

- Он должен быть `std::weakly_incrementable`, что означает поддержку оператора инкремента `++` для выполнения итераций по последовательности. Все итераторы поддерживают этот оператор.
- Он также должен быть `std::indirectly_copyable`, что определяет связь между итератором копируемого диапазона `input_range` и итератором вывода. В частности, итератор диапазона `input_range` должен быть `std::indirectly_readable`, что позволяет алгоритму `copy` разыменовывать этот итератор для чтения элемента заданного типа, а итератор вывода должен быть `std::indirectly_writable`, что позволяет алгоритму `copy` разыменовывать этот итератор для записи скопированного элемента этого типа в целевой диапазон.

Концепты `indirectly_readable` и `indirectly_writable`, в свою очередь, имеют много дополнительных требований. В дальнейшем для удобства мы будем говорить «итератор вывода», когда алгоритму требуется итератор `weakly_incrementable`.

Используйте алгоритмы диапазонов C++20 вместо старых алгоритмов для обычных диапазонов. Передача в аргументе целого контейнера, а не итераторов `begin` и `end` упрощает ваш код и предотвращает случайное несовпадение итераторов, когда итератор `begin` указывает на один контейнер, а итератор `end` указывает на другой.

Алгоритм for_each

В этом примере алгоритм `for_each` из пространства имен `std::ranges` (C++20) используется дважды. Первый вызов в строке 17 увеличивает каждый элемент контейнера `values` на 2 и выводит результат.

```
15    // Умножаем каждый элемент на 2 и выводим результат
16    std::cout << "\nDisplay each element multiplied by two: ";
17    std::ranges::for_each(values, [](auto i) {std::cout << i * 2 << " ";});
18
```

```
Display each element multiplied by two: 2 4 6 8
```

Двумя аргументами алгоритма `std::ranges::for_each` в данном случае являются:

- диапазон `input_range` (`values`), содержащий элементы для обработки;
- функция с одним аргументом, которой разрешено изменять свой аргумент в `input_range`.

Алгоритм `for_each` многократно вызывает функцию, указанную во втором аргументе, каждый раз передавая ей по одному элементу из своего первого аргумента `input_range`.

Лямбда с пустым списком захвата

Лямбда-выражение в строке 17 умножает свой параметр `i` на `2` и отображает результат. Каждое лямбда-выражение начинается со списка захвата в квадратных скобках (`[]`), за которым следует список параметров и тело функции. В данном случае список захвата пуст, поэтому он не захватывает никаких локальных переменных. Тип параметра — `auto`, поэтому мы имеем универсальную лямбду, для которой компилятор автоматически определяет тип на основе контекста. Поскольку мы выполняем итерации по значениям `int`, компилятор определяет тип параметра `i` как `int`. Лямбда-выражение из строки 17, когда оно специализировано для значений `int`, похоже на автономную функцию:

```
void timesTwo(int i) {
   cout << i * 2 << " ";
}
```

Если бы мы определили эту функцию, мы могли бы передать ее алгоритму `for_each`:

```
std::ranges::for_each(values, timesTwo);
```

Захват локальных переменных в лямбда-выражении

В строке 21 вызывается алгоритм `for_each` для вычисления суммы всех элементов `values`. Список захвата `[&sum]` захватывает локальную переменную `sum` (определена в строке 20) по ссылке (`&`), поэтому лямбда-выражение может изменять значение этой переменной[1, 2, 3]. Алгоритм `for_each` передает каждый элемент контейнера `values` лямбда-выражению, которое прибавляет это число к сумме. В строке 22 отображается итоговое значение переменной `sum`. Без амперсанда переменная `sum` была бы захвачена по значению, что привело бы к ошибке компиляции, поскольку аргументы лямбды по умолчанию обрабатываются как константы.

---

[1] C++ Core Guidelines, «F.50: Use a Lambda When a Function Won't Do (to Capture Local Variables, or to Write a Local Function)». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-capture-vs-overload.

[2] C++ Core Guidelines, «F.52: Prefer Capturing By Reference in Lambdas That Will Be Used Locally, Including Passed to Algorithms». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-reference-capture.

[3] C++ Core Guidelines, «F.53: Avoid Capturing By Reference in Lambdas That Will Be Used Non-Locally, Including Returned, Stored on the Heap, or Passed to Another Thread». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-value-capture.

```
19    // Прибавляем каждый элемент к sum
20    int sum{0}; // Инициализируем sum нулем
21    std::ranges::for_each(values, [&sum](auto i) {sum += i;});
22    std::cout << "\nSum of value's elements is: " << sum << "\n";
23 }
```

```
Sum of value's elements is: 10
```

Списки захвата [&] и [=]

Список захвата лямбды может захватывать несколько переменных из области видимости внешней функции, если в нем перечислены эти переменные, разделенные запятыми. Вы также можете захватить сразу несколько переменных, используя список захвата `[&]` или `[=]`:

- Список захвата `[&]` указывает, что каждая переменная из внешней области видимости, используемая в теле лямбды, должна быть захвачена по ссылке.
- Список захвата `[=]` указывает, что каждая переменная из внешней области видимости, используемая в теле лямбды, должна быть захвачена по значению.

Впрочем, лучшей практикой будет явное указание в списке именно тех переменных, которые надо захватить. Каждая переменная в списке захвата, которой предшествует символ `&`, будет захвачена по ссылке.

Типы, возвращаемые лямбда-выражением

Компилятор может автоматически определить возвращаемый тип лямбды по имеющейся в ее теле инструкции `return`. Вы также можете явно указать возвращаемый тип лямбды, используя синтаксис конечного возвращаемого типа C++11:

```
[](списокПараметров) -> тип {телоЛямбды}
```

Конечный возвращаемый тип (`-> тип`) помещается между закрывающей правой скобкой списка параметров и телом лямбда-выражения.

14.4. Алгоритмы

В разделах 14.4.1–14.4.14 демонстрируются многие алгоритмы стандартной библиотеки C++. Большинство представленных нами алгоритмов — это их версии из пространства имен `std::ranges` (C++20).

14.4.1. Алгоритмы fill, fill_n, generate и generate_n

На илл. 14.2 показана работа с алгоритмами из пространства имен `std::ranges` (C++20): `fill`, `fill_n`, `generate` и `generate_n`.

- Алгоритмы `fill` и `fill_n` присваивают определенное значение каждому элементу или первым `n` элементам диапазона.

- Алгоритмы `generate` и `generate_n` используют функцию-генератор[1] для создания значений каждого элемента диапазона или первых `n` элементов диапазона. Эта функция возвращает значение и не принимает аргументы.

В строках 9–12 определена функция `nextLetter`, генерирующая буквы. Мы также реализуем это в виде лямбды, чтобы вы могли увидеть сходство[2]. В строке 15 определены символы: 10-элементный контейнер `std::array`, содержащий значения типа `char`, которые мы будем обрабатывать в этом примере.

```
 1 // fig14_02.cpp
 2 // Алгоритмы fill, fill_n, generate и generate_n
 3 #include <algorithm> // Определения алгоритмов
 4 #include <array> // Определение шаблона класса array
 5 #include <iostream>
 6 #include <iterator> // ostream_iterator
 7
 8 // Возвращаем следующую букву (начиная с A)
 9 char nextLetter() {
10    static char letter{'A'};
11    return letter++;
12 }
13
14 int main() {
15    std::array<char, 10> chars{};
```

Илл. 14.2 | Алгоритмы fill, fill_n, generate и generate_n

Алгоритм fill

В строке 16 используется алгоритм C++20 `std::ranges::fill` для записи символа `'5'` в каждый элемент массива `chars`. Первым аргументом алгоритма должен быть диапазон `output_range`, чтобы алгоритм мог:

- выполнить итерации по диапазону, увеличивая итератор оператором `++`;
- разыменовать итератор для записи нового значения в текущий элемент.

Диапазон `array` имеет непрерывные итераторы, которые поддерживают все итерационные операции. Таким образом, алгоритм `fill` может увеличивать итераторы оператором `++`. Кроме того, массив `chars` не объявлен как константа, поэтому алгоритм `fill` может разыменовывать итераторы для записи значений в диапазон. В строке 20 отображаются элементы массива `chars`. В выходных данных этого примера мы выделили жирным шрифтом изменения, произведенные алгоритмами в массиве.

[1] Не следует путать с корутиной-генератором (C++20), с которой вы встретитесь в главе 18.

[2] Функция nextLetter не является потокобезопасной. В главе 17 обсуждается создание потокобезопасных функций.

```
16    std::ranges::fill(chars, '5'); // Заполняем массив chars значением 5
17
18    std::cout << "chars after filling with 5s: ";
19    std::ostream_iterator<char> output{std::cout, " "};
20    std::ranges::copy(chars, output);
21
```

```
chars after filling with 5s: 5 5 5 5 5 5 5 5 5 5
```

Алгоритм fill_n

В строке 23 используется алгоритм C++20 `std::ranges::fill_n` для размещения символа `'A'` (третий аргумент) в первых пяти элементах массива `chars` (второй аргумент). Первым аргументом должен быть как минимум итератор вывода. Объект `array` поддерживает непрерывные итераторы, и массив не объявлен как константа, поэтому вызов `chars.begin()` возвращает итератор, который `fill_n` может использовать для записи значений в массив.

```
22    // Заполняем первые пять элементов chars символом 'A'
23    std::ranges::fill_n(chars.begin(), 5, 'A');
24
25    std::cout << "\nchars after filling five elements with 'A's: ";
26    std::ranges::copy(chars, output);
27
```

```
chars after filling five elements with 'A's: A A A A A 5 5 5 5 5
```

Алгоритм generate

В строке 29 используется алгоритм C++20 `std::ranges::generate` для генерации значений всех элементов массива `chars`. Первым аргументом должен быть диапазон `output_range`. Второй аргумент — это функция, которая не принимает аргументы и возвращает значение. В функции `nextLetter` (строки 9–12) определяется статическая локальная переменная `letter` типа `char`, инициализированная значением `'A'`. В строке 11 возвращается текущее значение переменной `letter`, после чего переменная увеличивается для использования в следующем вызове функции.

```
28    // Генерируем значения всех элементов chars функцией nextLetter
29    std::ranges::generate(chars, nextLetter);
30
31    std::cout << "\nchars after generating letters A-J: ";
32    std::ranges::copy(chars, output);
33
```

```
chars after generating letters A-J: A B C D E F G H I J
```

Алгоритм generate_n

В строке 35 используется алгоритм C++20 `std::ranges::generate_n` для размещения результатов вызовов функции `nextLetter` в пяти элементах массива, начиная

с chars.begin(). Первым аргументом должен быть как минимум итератор ввода или вывода (input_or_output_iterator), который является indirectly_writable, чтобы алгоритм мог разыменовать его для записи в указанный диапазон.

```
34    // Генерируем значения первых пяти элементов chars функцией nextLetter
35    std::ranges::generate_n(chars.begin(), 5, nextLetter);
36
37    std::cout << "\nchars after generating K-O into elements 0-4: ";
38    std::ranges::copy(chars, output);
39
```

```
chars after generating K-O into elements 0-4: K L M N O F G H I J
```

Использование алгоритма generate_n с лямбдой

В строках 41–46 используется алгоритм C++20 std::ranges::generate_n и лямбда-выражение без аргументов (строки 42–45), возвращающее сгенерированную букву. Компилятор автоматически определяет по инструкции return возвращаемый тип char.

```
40    // Генерируем значения первых трех элементов chars лямбдой
41    std::ranges::generate_n(chars.begin(), 3,
42       [](){ // Лямбда, не принимающая аргументы
43          static char letter{'A'};
44          return letter++;
45       }
46    );
47
48    std::cout << "\nchars after generating A-C into elements 0-2: ";
49    std::ranges::copy(chars, output);
50    std::cout << "\n";
51 }
```

```
chars after generating A-C into elements 0-2: A B C N O F G H I J
```

14.4.2. Алгоритмы equal, mismatch и lexicographical_compare

На илл. 14.3 показано сравнение последовательностей значений на равенство с использованием алгоритмов equal, mismatch и lexicographical_compare из пространства имен std::ranges C++20. В строках 12–14 создаются и инициализируются три массива, а в строках 17–22 отображается их содержимое.

```
1 // fig14_03.cpp
2 // Алгоритмы equal, mismatch и lexicographical_compare
3 #include <algorithm> // Определения алгоритмов
4 #include <array> // Определение шаблона класса array
5 #include <fmt/format.h> // Или в C++20: #include <format>
6 #include <iomanip>
7 #include <iostream>
```

```
 8 #include <iterator> // ostream_iterator
 9 #include <string>
10
11 int main() {
12    std::array a1{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
13    std::array a2{a1}; // Инициализируем массив a2 копией массива a1
14    std::array a3{1, 2, 3, 4, 1000, 6, 7, 8, 9, 10};
15    std::ostream_iterator<int> output{std::cout, " "};
16
17    std::cout << "a1 contains: ";
18    std::ranges::copy(a1, output);
19    std::cout << "\na2 contains: ";
20    std::ranges::copy(a2, output);
21    std::cout << "\na3 contains: ";
22    std::ranges::copy(a3, output);
23
```

```
a1 contains: 1 2 3 4 5 6 7 8 9 10
a2 contains: 1 2 3 4 5 6 7 8 9 10
a3 contains: 1 2 3 4 1000 6 7 8 9 10
```

Илл. 14.3 | Алгоритмы equal, mismatch и lexicographical_compare

Алгоритм equal

В строках 26 и 30 используется алгоритм C++20 `std::ranges::equal` для сравнения двух `input_range` на предмет равенства. Алгоритм возвращает `false`, если последовательности отличаются по длине. Если их длина совпадает, он сравнивает соответствующие элементы каждого из диапазонов с помощью оператора `==`, возвращая `true`, если все элементы равны, или `false` в противном случае. В строке 26 сравниваются элементы массива `a1` с элементами массива `a2`. В этом примере они равны. В строке 30 сравниваются массивы `a1` и `a3`, которые не равны.

```
24    // Сравниваем массивы a1 и a2 на идентичность
25    std::cout << fmt::format("\n\na1 is equal to a2: {}\n",
26       std::ranges::equal(a1, a2));
27
28    // Сравниваем массивы a1 и a3 на идентичность
29    std::cout << fmt::format("a1 is equal to a3: {}\n",
30       std::ranges::equal(a1, a3));
31
```

```
a1 is equal to a2: true
a1 is equal to a3: false
```

Алгоритм equal с бинарной предикатной функцией

Многие алгоритмы стандартной библиотеки, сравнивающие элементы, позволяют вам указать функцию, определяющую способ сравнения элементов. Например, вы можете передать алгоритму `equal` функцию, принимающую в качестве аргумен-

тов два элемента для сравнения и возвращающую значение `bool` в зависимости от их равенства. Поскольку функция получает два аргумента и возвращает `bool`, она называется бинарной предикатной функцией. Это может быть полезно для диапазонов, содержащих объекты, для которых не определен оператор `==`, или объекты, содержащие указатели. Например, вы можете сравнивать объекты класса `Employee` по возрасту, ID или местоположению, а не сравнивать целые объекты. Вы можете сравнить объекты, на которые ссылаются указатели, вместо сравнения адресов, хранящихся в указателях.

Алгоритмы C++20 `std::ranges` также поддерживают **проекции (projections)**, позволяющие алгоритмам обрабатывать подмножества каждого объекта в диапазоне. Например, чтобы отсортировать объекты класса `Employee` по уровню зарплаты, вы можете использовать проекцию, содержащую зарплату каждого сотрудника. При сортировке порядок будет определен сравнением только уровня зарплаты и объекты класса `Employee` будут расположены требуемым образом. Мы выполним подобную сортировку в разделе 14.6.

Алгоритм mismatch

Алгоритм C++20 `std::ranges::mismatch` (строка 33) сравнивает два `input_range`. Алгоритм возвращает `std::ranges::mismatch_result`, содержащий итераторы `in1` и `in2`, указывающие на несовпадающие элементы в каждом диапазоне. Если все элементы совпадают, `in1` указывает на ограничитель первого диапазона, а `in2` — на ограничитель второго диапазона. В строке 33 автоматически определяется тип переменной `location`. Выражение в строке 35 находит индекс элемента, в котором обнаружено несоответствие:

```
location.in1 - a1.begin()
```

Это выражение вычисляется по количеству элементов между итераторами, как в арифметических операциях с указателями (глава 7). Подобно алгоритму `equal`, алгоритм `mismatch` может получать бинарную предикатную функцию для модификации операции сравнения (раздел 14.6).

```
32    // Ищем несоответствие между a1 и a3
33    auto location{std::ranges::mismatch(a1, a3)};
34    std::cout << fmt::format("a1 and a3 mismatch at index {} ({} vs. {})\n",
35       (location.in1 - a1.begin()), *location.in1, *location.in2);
36
```

```
a1 and a3 mismatch at index 4 (5 vs. 1000)
```

Возвращаемые типы и ключевое слово auto

Объявления типов шаблонов могут быть сложными и содержать незаметные с первого взгляда ошибки, особенно объявления возвращаемых типов алгоритмов стандартной библиотеки. Когда мы инициализируем переменную возвращаемым значением алгоритма (как в строке 33), мы используем `auto` для автоматическо-

го определения типа. Для пояснения рассмотрим возвращаемый тип алгоритма `std::ranges::mismatch` в строке 33:

```
std::ranges::mismatch_result<borrowed_iterator_t<R1>,
   borrowed_iterator_t<R2>>
```

Здесь `R1` и `R2` — это типы диапазонов, переданных алгоритму `mismatch`. В контексте объявлений массивов `a1` и `a3` возвращаемый тип алгоритма в строке 33 таков:

```
std::ranges::mismatch_result<borrowed_iterator_t<array<int, 10>>,
   borrowed_iterator_t<array<int, 10>>>
```

Как видите, гораздо удобнее написать одно ключевое слово `auto`, чтобы компилятор самостоятельно «сформулировал» это сложное объявление. Для автоматического определения типов аргументов также может использоваться CTAD (C++17). Поэтому мы можем объявить тип переменной очень просто: `std::ranges::mismatch_result`.

Алгоритм lexicographical_compare

Алгоритм C++20 `std::ranges::lexicographical_compare` (строки 41–42) сравнивает два диапазона `input_range`, в данном случае две строки типа `string`. Подобно контейнерам, объекты `string` содержат итераторы, позволяющие работать с ними как с диапазонами C++20. Алгоритм сравнивает диапазоны в цикле и возвращает `true`, если найдено отличие и элемент в первом диапазоне меньше, чем во втором. В противном случае алгоритм возвращает `false`. Он применяется для лексикографического упорядочивания последовательностей. Он также может получать бинарную предикатную функцию, возвращающую `true`, если ее первый аргумент следует считать меньшим, чем второй. Конечно, строки можно обрабатывать операторами равенства и сравнения, поэтому вызов алгоритма `lexicographical_compare` в строке 42 можно заменить на `s1 < s2`. Этот алгоритм в первую очередь предназначен для работы с диапазонами, которые нельзя непосредственно сравнивать друг с другом, такими как `std::span`.

```
37    std::string s1{"HELLO"};
38    std::string s2{"BYE BYE"};
39
40    // Выполняем лексикографическое сравнение двух строк
41    std::cout << fmt::format("\"{}\" < \"{}\": {}\n", s1, s2,
42       std::ranges::lexicographical_compare(s1, s2));
43 }
```

```
"HELLO" < "BYE BYE": false
```

14.4.3. Алгоритмы remove, remove_if, remove_copy и remove_copy_if

На илл. 14.4 демонстрируется процесс удаления значений из последовательности с помощью алгоритмов `remove`, `remove_if`, `remove_copy` и `remove_copy_if` из про-

странства имен `std::ranges` C++20[1]. В строке 9 создается вектор `init`, содержащий целочисленные значения, который мы будем использовать для инициализации других векторов в этом примере.

```
1 // fig14_04.cpp
2 // Алгоритмы remove, remove_if, remove_copy и remove_copy_if
3 #include <algorithm> // Определения алгоритмов
4 #include <iostream>
5 #include <iterator> // ostream_iterator
6 #include <vector>
7
8 int main() {
9    std::vector init{10, 2, 15, 4, 10, 6};
10   std::ostream_iterator<int> output{std::cout, " "};
11
```

Илл. 14.4 | Алгоритмы remove, remove_if, remove_copy и remove_copy_if

Алгоритм remove

В строках 12–14 вектор `v1` инициализируется копией вектора `init` и содержимое `v1` выводится на экран. В строке 17 используется алгоритм C++20 `std::ranges::remove` для удаления из `v1` всех элементов, равных `10`. Первым аргументом должен быть `forward_range`, поддерживающий однонаправленные итераторы. Эти итераторы должны быть `std::permutable`, то есть позволяющими выполнять замену или перемещение элементов, поскольку данный алгоритм удаляет элементы. Класс `vector` поддерживает более мощные итераторы произвольного доступа, поэтому может работать с любым алгоритмом, требующим итераторы категорией ниже. Этот алгоритм не уничтожает удаленные элементы. Вместо этого он помещает оставшиеся элементы в начало контейнера и возвращает поддиапазон, указывающий на недоступные элементы. Этот поддиапазон не следует использовать, его элементы обычно удаляются из контейнера, как мы покажем далее.

```
12    std::vector v1{init}; // Инициализируем вектор v1 копией вектора init
13    std::cout << "v1: ";
14    std::ranges::copy(v1, output);
15
16    // Удаляем из v1 элементы со значением 10
17    auto removed{std::ranges::remove(v1, 10)};
18    v1.erase(removed.begin(), removed.end());
19    std::cout << "\nv1 after removing 10s: ";
20    std::ranges::copy(v1, output);
21
```

```
v1: 10 2 15 4 10 6
v1 after removing 10s: 2 15 4 6
```

[1] В январе 2022 г. алгоритм std::ranges::remove не компилировался в последней версии компилятора clang++.

Идиома «erase — remove»

В строке 18 вызывается функция `erase` для удаления поддиапазона, представляющего недоступные элементы вектора. Это уменьшает размер вектора до оставшегося количества элементов. В строке 20 выводится обновленное содержимое `v1`.

Комбинация `remove` и `erase` в строках 17 и 18 выполняет операцию, известную как **идиома «erase — remove»**[1]. Вы удаляете элементы с помощью `remove` или `remove_if`, а потом вызываете функцию `erase` для уничтожения неиспользуемых элементов.

Алгоритмы `std::remove` и `std::remove_if` обычного диапазона возвращают каждый по одному итератору, указывающему на первый недействительный элемент в векторе. При использовании алгоритмов обычного диапазона вы можете выполнить идиому «erase — remove» одной инструкцией:

```
v1.erase(std::remove(v1.begin(), v1.end(), 10), v1.end());
```

К сожалению, такие функции контейнеров, как `erase`, пока не поддерживают диапазоны C++20.

Функции std::erase и std::erase_if (C++20)

Чтобы упростить идиому «erase — remove», в C++20 были добавлены функции `std::erase` и `std::erase_if`. Они перегружены в заголовках `<string>`, `<vector>`, `<deque>`, `<list>` и `<forward_list>`, а функция `std::erase_if` также перегружена в `<map>`, `<set>`, `<unordered_map>` и `<unordered_set>`. Каждая из этих функций выполняет идиому «erase — remove» в одном вызове, получая контейнер в качестве первого аргумента. Например, строки 17–18 могут быть заменены простой инструкцией:

```
std::erase(v1, 10);
```

Алгоритм remove_copy

В строке 28 используется алгоритм C++20 `std::ranges::remove_copy` для копирования в вектор `c1` всех элементов вектора `v2`, не равных `10`. Первым аргументом должен быть диапазон `input_range`, который поддерживает итераторы `input_iterator` для чтения элементов. Опять же, `vector` поддерживает более мощные итераторы произвольного доступа, поэтому соответствует минимальным требованиям алгоритма `remove_copy`. Второй аргумент мы рассматриваем ниже. В строке 30 выводятся все элементы вектора `c1`.

```
22    std::vector v2{init}; // Инициализируем вектор v2 копией вектора init
23    std::cout << "\n\nv2: ";
24    std::ranges::copy(v2, output);
25
26    // Копируем элементы из v2 в c1, удаляя элементы со значением 10
```

[1] «Erase–remove idiom». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Erase-remove_idiom.

```
27    std::vector<int> c1{};
28    std::ranges::remove_copy(v2, std::back_inserter(c1), 10);
29    std::cout << "\nc1 after copying v2 without 10s: ";
30    std::ranges::copy(c1, output);
31
```

```
v2: 10 2 15 4 10 6
c1 after copying v2 without 10s: 2 15 4 6
```

Адаптеры итератора: back_inserter, front_inserter и inserter

Итератор вывода во втором аргументе указывает, куда будут записаны скопированные элементы, — как правило, это другой контейнер. Алгоритм `remove_copy` не проверяет доступность нужного количества памяти в целевом контейнере для хранения всех скопированных элементов. Таким образом, итератор во втором аргументе должен ссылаться на контейнер, предоставляющий достаточно памяти для всех элементов. Без предварительного выделения памяти вы можете использовать адаптер итератора, расширяющий контейнер, чтобы вставлять больше элементов по мере добавления новых. В строке 28 в инструкции `std::back_inserter(c1)` используется адаптер итератора `back_inserter` (заголовок `<iterator>`) для создания итератора `back_insert_iterator`. Когда алгоритм использует этот итератор для вставки элемента в вектор `c1`, итератор вызывает функцию `push_back` контейнера, чтобы поместить элемент в конец вектора. Если контейнеру требуется больше памяти, он увеличивается, чтобы вместить новый элемент. Итератор `back_insert_iterator` несовместим с массивами `array`, поскольку они являются контейнерами фиксированного размера и не предоставляют функцию `push_back`.

Существуют два аналогичных адаптера:

- Адаптер `front_inserter` создает `front_insert_iterator`, использующий функцию `push_front` контейнера для вставки элемента в начало контейнера.
- Адаптер `inserter` создает `insert_iterator`, использующий функцию `insert` контейнера для вставки элемента из контейнера, указанного в первом аргументе адаптера, в позицию, указанную итератором во втором аргументе адаптера.

Алгоритм remove_if

В строке 38 используется алгоритм C++20 `std::ranges::remove_if` для удаления из вектора `v3` всех элементов, для которых унарная предикатная функция `greaterThan9` возвращает `true`. Первым аргументом алгоритма должен быть диапазон `forward_range`, который позволяет алгоритму `remove_if` считывать элементы. Унарная предикатная функция должна получить один параметр и возвратить значение `bool`.

```
32    std::vector v3{init}; // Инициализируем вектор v3 копией вектора init
33    std::cout << "\n\nv3: ";
34    std::ranges::copy(v3, output);
35
```

```
36    // Удаляем из v3 элементы со значением больше 9
37    auto greaterThan9{[](auto x) {return x > 9;}};
38    auto removed2{std::ranges::remove_if(v3, greaterThan9)};
39    v3.erase(removed2.begin(), removed2.end());
40    std::cout << "\nv3 after removing elements greater than 9: ";
41    std::ranges::copy(v3, output);
42
```

```
v3: 10 2 15 4 10 6
v3 after removing elements greater than 9: 2 4 6
```

В строке 37 определяется унарная предикатная функция в виде обобщенной лямбды:

```
[](auto x){return x > 9;}
```

Она возвращает `true`, если аргумент больше `9`; в противном случае — `false`. Компилятор использует автоматическое определение типа как для параметра лямбды, так и для возвращаемых типов:

- Вектор содержит целые числа, поэтому компилятор определяет тип параметра как `int`.
- Лямбда-выражение возвращает результат вычисления условия, поэтому автоматически определяет возвращаемый тип как `bool`.

Подобно алгоритму `remove`, алгоритм `remove_if` не изменяет количество элементов в контейнере. Вместо этого он перемещает в начало контейнера все элементы, которые не были удалены, и возвращает поддиапазон из недоступных элементов. Мы используем этот поддиапазон в строке 39, уничтожая недоступные элементы вектора `v3`. В строке 41 выводится обновленное содержимое вектора. Строки 38–39 могут быть заменены на функцию C++20 `std::erase_if`:

```
std::erase_if(v3, greaterThan9);
```

Алгоритм remove_copy_if

В строке 49 алгоритм C++20 `std::ranges::remove_copy_if` копирует из диапазона `input_range` (вектора `v4`) все элементы, для которых унарная предикатная функция (лямбда-выражение в строке 37) возвращает `true`. Второй аргумент должен быть итератором вывода, чтобы копируемые элементы могли быть записаны в конечный диапазон. Мы используем адаптер `back_inserter` для вставки скопированных элементов в вектор `c2`. В строке 51 выводится содержимое вектора `c2`.

```
43    std::vector v4{init}; // Инициализируем вектор v4 копией вектора init
44    std::cout << "\n\nv4: ";
45    std::ranges::copy(v4, output);
46
47    // Копируем элементы из v4 в c2, удаляя элементы со значением больше 9
48    std::vector<int> c2{};
49    std::ranges::remove_copy_if(v4, std::back_inserter(c2), greaterThan9);
```

```
50    std::cout << "\nc2 after copying v4 without elements greater than 9: ";
51    std::ranges::copy(c2, output);
52    std::cout << "\n";
53 }
```

```
v4: 10 2 15 4 10 6
c2 after copying v4 without elements greater than 9: 2 4 6
```

14.4.4. Алгоритмы replace, replace_if, replace_copy и replace_copy_if

На илл. 14.5 демонстрируется замена элементов в последовательности с помощью алгоритмов `replace`, `replace_if`, `replace_copy` и `replace_copy_if` из библиотеки C++20 `std::ranges`.

```
 1 // fig14_05.cpp
 2 // Алгоритмы replace, replace_if, replace_copy и replace_copy_if
 3 #include <algorithm>
 4 #include <array>
 5 #include <iostream>
 6 #include <iterator> // ostream_iterator
 7
 8 int main() {
 9    std::ostream_iterator<int> output{std::cout, " "};
10
```

Илл. 14.5 | Алгоритмы replace, replace_if, replace_copy и replace_copy_if

Алгоритм replace

В строке 16 алгоритм C++20 `std::ranges::replace` заменяет все числа `10` на `100` в диапазоне `input_range` (`a1`). Первый аргумент должен поддерживать итераторы `indirectly_writable`, чтобы алгоритм `replace` мог разыменовывать эти итераторы для замены значений в диапазоне. Массив `a1` не объявлен как константа, поэтому его итераторы можно использовать для записи.

```
11    std::array a1{10, 2, 15, 4, 10, 6};
12    std::cout << "a1: ";
13    std::ranges::copy(a1, output);
14
15    // Заменяем в a1 все значения 10 на 100
16    std::ranges::replace(a1, 10, 100);
17    std::cout << "\na1 after replacing 10s with 100s: ";
18    std::ranges::copy(a1, output);
19
```

```
a1: 10 2 15 4 10 6
a1 after replacing 10s with 100s: 100 2 15 4 100 6
```

Алгоритм replace_copy

В строке 26 алгоритм C++20 `std::ranges::replace_copy` копирует все элементы диапазона `input_range (a2)`, заменяя при этом каждое число `10` на `100`. Второй аргумент должен быть итератором вывода, представляющим позицию для записи скопированных элементов. Алгоритм копирует или заменяет каждый элемент диапазона `input_range`, поэтому мы определили массив `c1` с таким же количеством элементов, как в массиве `a2`. Но мы могли бы использовать и пустой вектор в сочетании с `back_inserter`.

```
20    std::array a2{10, 2, 15, 4, 10, 6};
21    std::array<int, a2.size()> c1{};
22    std::cout << "\n\na2: ";
23    std::ranges::copy(a2, output);
24
25    // Копируем элементы из a2 в c1, заменяя все значения 10 на 100
26    std::ranges::replace_copy(a2, c1.begin(), 10, 100);
27    std::cout << "\nc1 after replacing a2's 10s with 100s: ";
28    std::ranges::copy(c1, output);
29
```

```
a2: 10 2 15 4 10 6
c1 after replacing a2's 10s with 100s: 100 2 15 4 100 6
```

Алгоритм replace_if

В строке 36 алгоритм C++20 `std::ranges::replace_if` заменяет каждый элемент диапазона `input_range (a3)`, для которого унарная предикатная функция (`greaterThan9` в строке 35) возвращает `true`. Первый аргумент должен поддерживать итераторы вывода, чтобы алгоритм `replace_if` мог заменять значения в диапазоне. Здесь мы заменяем каждое число, большее `9`, на `100`.

```
30    std::array a3{10, 2, 15, 4, 10, 6};
31    std::cout << "\n\na3: ";
32    std::ranges::copy(a3, output);
33
34    // Заменяем в a3 все значения больше 9 на 100
35    auto greaterThan9{[](auto x) {return x > 9;}};
36    std::ranges::replace_if(a3, greaterThan9, 100);
37    std::cout << "\na3 after replacing values greater than 9 with 100s: ";
38    std::ranges::copy(a3, output);
39
```

```
a3: 10 2 15 4 10 6
a3 after replacing values greater than 9 with 100s: 100 2 100 4 100 6
```

Алгоритм replace_copy_if

В строке 46 алгоритм C++20 `std::ranges::replace_copy_if` копирует все элементы диапазона `input_range (a4)` и, если унарная предикатная функция возвращает `true`, заменяет их. Все числа больше `9` заменяются на `100`. Скопированные или

замененные элементы помещаются в массив c2 с позиции c2.begin(), которая должна быть итератором вывода.

```
40    std::array a4{10, 2, 15, 4, 10, 6};
41    std::array<int, a4.size()> c2{};
42    std::cout << "\n\na4: ";
43    std::ranges::copy(a4, output);
44
45    // Копируем элементы из a4 в c2, заменяя все значения больше 9 на 100
46    std::ranges::replace_copy_if(a4, c2.begin(), greaterThan9, 100);
47    std::cout << "\nc2 after replacing a4's values "
48       << "greater than 9 with 100s: ";
49    std::ranges::copy(c2, output);
50    std::cout << "\n";
51 }
```

```
a4: 10 2 15 4 10 6
c2 after replacing a4's values greater than 9 with 100s: 100 2 100 4 100 6
```

14.4.5. Алгоритмы перемешивания, подсчетов, поиска минимального и максимального элементов

На илл. 14.6 показана работа алгоритмов shuffle, count, count_if, min_element, max_element и minmax_element из пространства имен std::ranges C++20. Мы также используем алгоритм transform для возведения в степень чисел из диапазона.

```
 1 // fig14_06.cpp
 2 // Алгоритмы перемешивания, подсчетов, поиска минимального и максимального
      элементов
 3 #include <algorithm>
 4 #include <array>
 5 #include <iostream>
 6 #include <iterator>
 7 #include <random>
 8
 9 int main() {
10    std::array a1{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
11    std::ostream_iterator<int> output{std::cout, " "};
12
13    std::cout << "a1: ";
14    std::ranges::copy(a1, output);
15
```

```
a1: 1 2 3 4 5 6 7 8 9 10
```

Илл. 14.6 | Математические алгоритмы стандартной библиотеки

Алгоритм shuffle

В строке 18 алгоритм C++20 std::ranges::shuffle изменяет случайным образом порядок элементов массива a1. Для этого требуется диапазон random_access_range

с итераторами произвольного доступа, поэтому алгоритм можно использовать с контейнерами `array`, `vector` и `deque` (а также с традиционными массивами в стиле C). В объявлении алгоритма указано, что итераторы диапазона должны быть `permutable`, то есть допускать замену и перемещение элементов. Итераторы массива, не объявленного как константа, разрешают такие операции. Вторым аргументом алгоритма `shuffle` является генератор случайных чисел C++11 (раздел 5.8). В строке 20 отображается «перемешанный» массив.

```
16    // Инициализируем генератор случайных чисел для перемешивания элементов
         массива a1
17    std::default_random_engine randomEngine{std::random_device{}()};
18    std::ranges::shuffle(a1, randomEngine); // Перемешиваем в случайном порядке
19    std::cout << "\na1 shuffled: ";
20    std::ranges::copy(a1, output);
21
```

```
a1 shuffled: 5 4 7 6 3 9 1 8 10 2
```

Алгоритм count

В строке 27 алгоритм C++20 `std::ranges::count` подсчитывает количество элементов в диапазоне `a2`, содержащих значение `8`. Первым аргументом этого алгоритма должен быть диапазон `input_range`, чтобы алгоритм мог считывать из него элементы.

```
22    std::array a2{100, 2, 8, 1, 50, 3, 8, 8, 9, 10};
23    std::cout << "\n\na2: ";
24    std::ranges::copy(a2, output);
25
26    // Подсчитываем в массиве a2 количество элементов со значением 8
27    auto result1{std::ranges::count(a2, 8)};
28    std::cout << "\nCount of 8s in a2: " << result1;
29
```

```
a2: 100 2 8 1 50 3 8 8 9 10
Count of 8s in a2: 3
```

Алгоритм count_if

В строке 31 алгоритм C++20 `std::ranges::count_if` подсчитывает количество элементов в диапазоне `input_range` (`a2`), для которых унарная предикатная функция возвращает `true`. В этом примере мы снова используем унарную предикатную функцию в виде лямбды, возвращающей `true` для чисел больше `9`.

```
30    // Подсчитываем в массиве a2 количество элементов со значением больше 9
31    auto result2{std::ranges::count_if(a2, [](auto x){return x > 9;})};
32    std::cout << "\nCount of a2 elements greater than 9: " << result2;
33
```

```
Count of a2 elements greater than 9: 3
```

Алгоритм min_element

В строке 35 алгоритм C++20 `std::ranges::min_element` находит наименьший элемент в диапазоне `forward_range` (`a2`). Этот диапазон поддерживает однонаправленные итераторы, позволяющие алгоритму `min_element` считывать из него элементы. Алгоритм возвращает итератор, указывающий на первое вхождение наименьшего элемента диапазона, или ограничитель (следующую позицию после последнего элемента диапазона), если диапазон пуст. Как и во многих других алгоритмах, сравнивающих элементы, вы можете предоставить бинарную предикатную функцию, определяющую способ сравнения элементов и возвращающую `true`, если первый аргумент следует считать меньшим, чем второй. Перед разыменованием такого итератора необходимо убедиться, что он не указывает на ограничитель диапазона, как это сделано в строке 35. Вы должны обязательно выполнять такую проверку для тех итераторов, которые иногда могут указывать на ограничитель.

```
34    // Находим в массиве a2 минимальный элемент
35    if (auto result{std::ranges::min_element(a2)}; result != a2.end()) {
36       std::cout << "\n\na2 minimum element: " << *result;
37    }
38
```

```
a2 minimum element: 1
```

Алгоритм max_element

В строке 40 алгоритм C++20 `std::ranges::max_element` находит наибольший элемент в диапазоне `forward_range` (`a2`). Этот диапазон поддерживает однонаправленные итераторы, позволяющие алгоритму `max_element` считывать из него элементы. Алгоритм возвращает итератор, указывающий на первое вхождение наибольшего элемента диапазона, или ограничитель диапазона, если диапазон пуст. Вы можете предоставить бинарную предикатную функцию, указывающую способ сравнения элементов.

```
39    // Находим в массиве a2 максимальный элемент
40    if (auto result{std::ranges::max_element(a2)}; result != a2.end()) {
41       std::cout << "\na2 maximum element: " << *result;
42    }
43
```

```
a2 maximum element: 100
```

Алгоритм minmax_element

В строке 45 алгоритм C++20 `std::ranges::minmax_element` находит наименьший и наибольший элементы в диапазоне `forward_range` (`a2`). Алгоритм возвращает `std::ranges::minmax_element_result`, содержащий итераторы `min` и `max`, указывающие на наименьший и наибольший элементы соответственно:

- если в диапазоне есть несколько наименьших элементов, первый итератор будет указывать на первое вхождение наименьшего элемента;
- если в диапазоне присутствуют несколько наибольших элементов, второй итератор будет указывать на последнее вхождение наибольшего элемента — обратите внимание на это отличие от алгоритма `max_element`.

При работе с этим алгоритмом вы тоже можете предоставить бинарную предикатную функцию для сравнения элементов.

```
44    // Находим в массиве a2 минимальный и максимальный элементы
45    auto [min, max]{std::ranges::minmax_element(a2)};
46    std::cout << "\na2 minimum and maximum elements: "
47       << *min << " and " << *max;
48
```

```
a2 minimum and maximum elements: 1 and 100
```

Структурное связывание (C++17)

В строке 45 используется `minmax_element_result`, возвращаемый алгоритмом `minmax_element`, для инициализации переменных `min` и `max` с помощью объявления **структурного связывания (structured binding)**[1,2]. Это иногда называется распаковкой элементов и используется для извлечения в отдельные переменные элементов традиционного массива либо объекта `std::array`, `std::pair` или `std::tuple` (размер которых известен при компиляции). Структурное связывание также можно использовать для распаковки открытых элементов данных структуры или объекта класса. В тех примерах главы 13, где функция возвращала `std::pair`, мы могли использовать структурное связывание для распаковки.

Алгоритм transform

В строках 51–52 алгоритм C++20 `std::ranges::transform` преобразует элементы диапазона `input_range (a1)` в новые значения. Каждое новое значение записывается в заданный во втором аргументе алгоритма диапазон, который должен быть итератором вывода и может указывать на тот же контейнер, что и `input_range`, или на другой контейнер. Функция, предоставленная в качестве третьего аргумента, получает значение и возвращает новое. Порядок вызова этой функции для элементов диапазона не гарантируется, поэтому она не должна изменять элементы `input_range`. Чтобы изменить элементы исходного диапазона, используйте алгоритм `std::foreach`[3].

---

[1] «Structured Binding Declaration». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/language/structured_binding.

[2] Доминик Бернер (Dominik Berner), «Quick and Easy Unpacking in C++ with Structured Bindings», 24 мая 2018 г. Был доступен: 15 сентября 2023 г. https://dominikberner.ch/structured-bindings/.

[3] «Algorithms Library — mutating Sequence Operations — Transform». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/alg.transform.

```
49    // Возводим в куб каждый элемент массива a1; помещаем результаты в cubes
50    std::array<int, a1.size()> cubes{};
51    std::ranges::transform(a1, cubes.begin(),
52       [](auto x){return x * x * x;});
53    std::cout << "\n\na1 values cubed: ";
54    std::ranges::copy(cubes, output);
55    std::cout << "\n";
56 }
```

```
a1 values cubed: 125 64 343 216 27 729 1 512 1000 8
```

Перегруженная версия алгоритма `transform` принимает два диапазона `input_range`, итератор вывода и функцию, принимающую два аргумента и возвращающую результат. Эта версия передает соответствующие элементы из каждого диапазона `input_range` в функцию и выводит результаты через итератор вывода.

14.4.6. Алгоритмы поиска и сортировки

На илл. 14.7 демонстрируется работа некоторых базовых алгоритмов поиска и сортировки: `find`, `find_if`, `sort`, `binary_search`, `all_of`, `any_of`, `none_of` и `find_if_not` из пространства имен `std::ranges` C++20.

```
 1 // fig14_07.cpp
 2 // Алгоритмы поиска и сортировки стандартной библиотеки
 3 #include <algorithm> // Определения алгоритмов
 4 #include <array> // Определение шаблона класса array
 5 #include <iostream>
 6 #include <iterator>
 7
 8 int main() {
 9    std::array values{10, 2, 17, 5, 16, 8, 13, 11, 20, 7};
10    std::ostream_iterator<int> output{std::cout, " "};
11
12    std::cout << "values contains: ";
13    std::ranges::copy(values, output); // Выводим элементы values
14
```

```
values contains: 10 2 17 5 16 8 13 11 20 7
```

Илл. 14.7 | Алгоритмы поиска и сортировки стандартной библиотеки

Алгоритм find

Алгоритм C++20 `std::ranges::find` (строка 16) выполняет линейный поиск $O(n)$ определенного значения (`16`) в диапазоне `input_range` (`values`). Алгоритм возвращает итератор, указывающий на первое вхождение элемента, содержащего искомое значение, или ограничитель диапазона, если значение не найдено (строки 24–29). Мы используем возвращенный итератор в строке 17 для вычисления индекса элемента, содержащего искомое значение.

```
15    // Ищем в массиве values первое вхождение числа 16
16    if (auto loc1{std::ranges::find(values, 16)}; loc1 != values.cend()) {
17       std::cout << "\n\nFound 16 at index: " << (loc1 - values.cbegin());
18    }
19    else { // Если число 16 не найдено
20       std::cout << "\n\n16 not found";
21    }
22
23    // Ищем в массиве values первое вхождение числа 100
24    if (auto loc2{std::ranges::find(values, 100)}; loc2 != values.cend()) {
25       std::cout << "\nFound 100 at index: " << (loc2 - values.cbegin());
26    }
27    else { // Если число 100 не найдено
28       std::cout << "\n100 not found";
29    }
30
```

```
Found 16 at index: 4
100 not found
```

Алгоритм find_if

Алгоритм C++20 `std::ranges::find_if` (строка 35) выполняет линейный поиск $O(n)$ первого элемента диапазона `input_range` (`values`), для которого унарная предикатная функция (`isGreaterThan10` из строки 32) возвращает `true`. Алгоритм возвращает итератор, указывающий на первое вхождение элемента, содержащего искомое значение, или ограничитель диапазона, если значение не найдено.

```
31    // Создаем переменную для хранения лямбды, чтобы повторно ее использовать
32    auto isGreaterThan10{[](auto x){return x > 10;}};
33
34    // Ищем в массиве values первое вхождение числа больше 10
35    auto loc3{std::ranges::find_if(values, isGreaterThan10)};
36
37    if (loc3 != values.cend()) { // Если найдено число больше 10
38       std::cout << "\n\nFirst value greater than 10: " << *loc3
39          << "\nFound at index: " << (loc3 - values.cbegin());
40    }
41    else { // Если число больше 10 не найдено
42       std::cout << "\n\nNo values greater than 10 were found";
43    }
44
```

```
First value greater than 10: 17
found at index: 2
```

Алгоритм sort

Алгоритм C++20 `std::ranges::sort` (строка 46) выполняет сортировку $O(n \log n)$, упорядочивая элементы диапазона `values` в порядке возрастания. Аргумент, обозначающий диапазон, должен быть `random_access_range`, то есть поддерживать

итераторы произвольного доступа, поэтому алгоритм `sort` может работать с контейнерами `array`, `vector` и `deque`, а также с традиционными массивами. Алгоритм может получать бинарную предикатную функцию, принимающую два аргумента и возвращающую значение `bool`, указывающее на порядок сортировки. Функция сравнивает два значения из сортируемой последовательности. Если предикат возвращает `true`, два элемента уже находятся в нужном порядке; в противном случае два элемента необходимо упорядочить.

```
45    // Сортируем элементы массива values
46    std::ranges::sort(values);
47    std::cout << "\n\nvalues after sort: ";
48    std::ranges::copy(values, output);
49
```

```
values after sort: 2 5 7 8 10 11 13 16 17 20
```

Алгоритм binary_search

Алгоритм C++20 `std::ranges::binary_search` (строка 51) выполняет бинарный поиск $O(\log n)$, определяя, находится ли определенное значение (`13`) в диапазоне `forward_range` (`values`). Этот диапазон должен быть отсортирован в порядке возрастания. Алгоритм возвращает значение `bool`, указывающее, было ли значение найдено в последовательности. В строке 59 демонстрируется вызов алгоритма `binary_search`, который не находит искомого значения. Этот алгоритм также может получать бинарную предикатную функцию, принимающую два аргумента и возвращающую значение `bool`. Функция возвращает `true`, если два сравниваемых элемента находятся в отсортированном порядке. Если вам нужно знать позицию искомого значения в контейнере, используйте алгоритмы `lower_bound` или `find` вместо `binary_search`.

```
50    // Вызываем binary search, чтобы узнать, есть ли в values число 13
51    if (std::ranges::binary_search(values, 13)) {
52       std::cout << "\n\n13 was found in values";
53    }
54    else {
55       std::cout << "\n\n13 was not found in values";
56    }
57
58    // Вызываем binary_search, чтобы узнать, есть ли в values число 100
59    if (std::ranges::binary_search(values, 100)) {
60       std::cout << "\n100 was found in values";
61    }
62    else {
63       std::cout << "\n100 was not found in values";
64    }
65
```

```
13 was found in values
100 was not found in values
```

Алгоритм all_of

Алгоритм C++20 `std::ranges::all_of` (строка 67) выполняет линейный поиск $O(n)$ и определяет, возвращает ли бинарная предикатная функция из второго аргумента (в данном случае лямбда-выражение `isGreaterThan10`) значение `true` для всех элементов первого аргумента: диапазона `input_range (values)`. Если это так, алгоритм `all_of` возвращает `true`; в противном случае он возвращает `false`.

```
66    // Выясняем, все ли числа в values больше 10
67    if (std::ranges::all_of(values, isGreaterThan10)) {
68       std::cout << "\n\nAll values elements are greater than 10";
69    }
70    else {
71       std::cout << "\n\nSome values elements are not greater than 10";
72    }
73
```

```
Some values elements are not greater than 10
```

Алгоритм any_of

Алгоритм C++20 `std::ranges::any_of` (строка 75) выполняет линейный поиск $O(n)$ и определяет, возвращает ли унарная предикатная функция из второго аргумента (в данном случае лямбда-выражение `isGreaterThan10`) значение `true` по крайней мере для одного элемента первого аргумента: диапазона `input_range (values)`. Если это так, алгоритм `any_of` возвращает `true`; в противном случае он возвращает `false`.

```
74    // Выясняем, есть ли в values числа больше 10
75    if (std::ranges::any_of(values, isGreaterThan10)) {
76       std::cout << "\n\nSome values elements are greater than 10";
77    }
78    else {
79       std::cout << "\n\nNo values elements are greater than 10";
80    }
81
```

```
Some values elements are greater than 10
```

Алгоритм none_of

Алгоритм C++20 `std::ranges::none_of` (строка 83) выполняет линейный поиск $O(n)$ и определяет, возвращает ли унарная предикатная функция из второго аргумента (в данном случае лямбда-выражение `isGreaterThan10`) значение `false` для всех элементов первого аргумента: диапазона `input_range (values)`. Если это так, алгоритм `none_of` возвращает `true`; в противном случае он возвращает `false`.

```
82    // Выясняем, все ли числа в values меньше 10
83    if (std::ranges::none_of(values, isGreaterThan10)) {
84       std::cout << "\n\nNo values elements are greater than 10";
85    }
86    else {
87       std::cout << "\n\nSome values elements are greater than 10";
88    }
89
```

```
Some values elements are greater than 10
```

Алгоритм find_if_not

Алгоритм C++20 `std::ranges::find_if_not` (строка 91) выполняет линейный поиск $O(n)$ и находит первое значение диапазона `input_range (values)`, для которого унарная предикатная функция (лямбда-выражение `isGreaterThan10`) возвращает значение `false`. Алгоритм возвращает итератор, указывающий на первое вхождение элемента, содержащего искомое значение, или ограничитель диапазона, если значение не найдено.

```
90    // Ищем первое вхождение числа не больше 10
91    auto loc4{std::ranges::find_if_not(values, isGreaterThan10)};
92
93    if (loc4 != values.cend()) { // Если найдено число не больше 10
94       std::cout << "\n\nFirst value not greater than 10: " << *loc4
95         << "\nfound at index: " << (loc4 - values.cbegin());
96    }
97    else { // Если число не больше 10 не найдено
98       std::cout << "\n\nOnly values greater than 10 were found";
99    }
100
101    std::cout << "\n";
102 }
```

```
First value not greater than 10: 2
found at index: 0
```

14.4.7. Алгоритмы swap, iter_swap и swap_ranges

На илл. 14.8 показана работа алгоритмов замены элементов `swap` и `iter_swap` из пространства имен `std` и алгоритма `swap_ranges` из пространства имен `std::ranges` C++20.

```
1 // fig14_08.cpp
2 // Алгоритмы swap, iter_swap и swap_ranges
3 #include <algorithm>
4 #include <array>
5 #include <iostream>
```

```
 6 #include <iterator>
 7
 8 int main() {
 9    std::array values{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
10    std::ostream_iterator<int> output{std::cout, " "};
11
12    std::cout << "values contains: ";
13    std::ranges::copy(values, output);
14
```

```
values contains: 1 2 3 4 5 6 7 8 9 10
```

Илл. 14.8 | Алгоритмы swap, iter_swap и swap_ranges

Алгоритм swap

В строке 15 алгоритм `std::swap` выполняет обмен значений двух аргументов. Обратите внимание: это не алгоритм диапазона или обычного диапазона. Функция получает ссылки на два значения, которые надо поменять местами. В данном случае мы передаем ей ссылки на первый и второй элементы массива.

```
15    std::swap(values[0], values[1]); // Меняем местами элементы с индексами 0 и 1
16
17    std::cout << "\nafter std::swap of values[0] and values[1]: ";
18    std::ranges::copy(values, output);
19
```

```
after std::swap of values[0] and values[1]: 2 1 3 4 5 6 7 8 9 10
```

Алгоритм iter_swap

В строке 21 алгоритм `std::iter_swap` выполняет обмен значений двух элементов, обозначенных однонаправленными итераторами обычного диапазона. Эти итераторы могут ссылаться на любые два элемента одного и того же типа.

```
20    // Используем итераторы, чтобы поменять местами элементы с индексами 0 и 1
21    std::iter_swap(values.begin(), values.begin() + 1);
22    std::cout << "\nafter std::iter_swap of values[0] and values[1]: ";
23    std::ranges::copy(values, output);
24
```

```
after std::iter_swap of values[0] and values[1]: 1 2 3 4 5 6 7 8 9 10
```

Алгоритм swap_ranges

В строке 31 алгоритм C++20 `std::ranges::swap_ranges` выполняет обмен значений двух диапазонов `input_range`. Если диапазоны имеют разную длину, алгоритм заменяет более короткую последовательность соответствующими элементами из более длинной последовательности. Диапазоны также должны поддерживать итераторы `indirectly_swappable`, чтобы алгоритм мог разыменовывать итераторы для замены соответствующих элементов в каждом диапазоне.

```
25    // Меняем местами содержимое массивов values и values2
26    std::array values2{10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
27    std::cout << "\n\nBefore swap_ranges\nvalues contains: ";
28    std::ranges::copy(values, output);
29    std::cout << "\nvalues2 contains: ";
30    std::ranges::copy(values2, output);
31    std::ranges::swap_ranges(values, values2);
32    std::cout << "\n\nAfter swap_ranges\nvalues contains: ";
33    std::ranges::copy(values, output);
34    std::cout << "\nvalues2 contains: ";
35    std::ranges::copy(values2, output);
36
```

```
Before swap_ranges
values contains: 1 2 3 4 5 6 7 8 9 10
values2 contains: 10 9 8 7 6 5 4 3 2 1

After swap_ranges
values contains: 10 9 8 7 6 5 4 3 2 1
values2 contains: 1 2 3 4 5 6 7 8 9 10
```

В строках 38–39 вызывается перегруженная версия алгоритма C++20 `std::ranges::swap_ranges`, позволяющая указать фрагменты двух диапазонов `input_range` для обмена значений. Здесь мы меняем местами первые пять элементов `values` и первые пять элементов `values2`, указывая диапазоны парами итераторов:

- `values.begin()` и `values.begin() + 5` обозначают первые пять элементов массива `values`;
- `values2.begin()` и `values2.begin() + 5` обозначают первые пять элементов массива `values2`.

Указание парами итераторов также работает с версией алгоритма для обычных диапазонов из пространства имен `std`. В этом примере два диапазона находятся в разных контейнерах, но диапазоны могут находиться и в одном контейнере, при этом они не должны пересекаться.

```
37    // Меняем местами первые пять элементов массивов values и values2
38    std::ranges::swap_ranges(values.begin(), values.begin() + 5,
39       values2.begin(), values2.begin() + 5);
40
41    std::cout << "\n\nAfter swap_ranges for 5 elements"
42       << "\nvalues contains: ";
43    std::ranges::copy(values, output);
44    std::cout << "\nvalues2 contains: ";
45    std::ranges::copy(values2, output);
46    std::cout << "\n";
47 }
```

```
After swap_ranges for 5 elements
values contains: 1 2 3 4 5 5 4 3 2 1
values2 contains: 10 9 8 7 6 6 7 8 9 10
```

14.4.8. Алгоритмы copy_backward, merge, unique, reverse, copy_if и copy_n

На илл. 14.9 демонстрируется работа алгоритмов `copy_backward`, `merge`, `unique`, `reverse`, `copy_if` и `copy_n` из пространства имен `std::ranges` C++20[1].

```
 1 // fig14_09.cpp
 2 // Алгоритмы copy_backward, merge, unique, reverse, copy_if и copy_n
 3 #include <algorithm>
 4 #include <array>
 5 #include <iostream>
 6 #include <iterator>
 7 #include <vector>
 8
 9 int main() {
10    std::array a1{1, 3, 5, 7, 9};
11    std::array a2{2, 4, 5, 7, 9};
12    std::ostream_iterator<int> output{std::cout, " "};
13
14    std::cout << "array a1 contains: ";
15    std::ranges::copy(a1, output); // Выводим массив a1
16    std::cout << "\narray a2 contains: ";
17    std::ranges::copy(a2, output); // Выводим массив a2
18
```

```
array a1 contains: 1 3 5 7 9
array a2 contains: 2 4 5 7 9
```

Илл. 14.9 | Алгоритмы copy_backward, merge, unique, reverse, copy_if и copy_n

Алгоритм copy_backward

Алгоритм C++20 `std::ranges::copy_backward` (строка 21) копирует диапазон `bidirectional_range` (первый аргумент, `a1`) в диапазон, указанный итератором вывода (второй аргумент). В данном случае итератор указывает на конец принимающего контейнера: `results.end()`. Алгоритм копирует элементы в обратном порядке и помещает их в контейнер, начиная с элемента, расположенного перед элементом `results.end()`, продвигаясь к началу контейнера. Алгоритм возвращает `std::ranges::copy_backward_result`, содержащий два итератора:

- первый указывает на `a1.end()`;
- второй указывает на последний скопированный элемент в принимающем диапазоне, то есть на начало `results`, поскольку копирование выполняется в обратном направлении.

[1] В январе 2022 г. алгоритм std::ranges::unique не компилировался в последней версии компилятора clang++.

Хотя элементы копируются в обратном порядке, они размещены в `results` в том же порядке, что и в массиве `a1`. Единственное различие между алгоритмами `copy` и `copy_backward`:

- итератор, возвращенный алгоритмом `copy`, указывает на следующий элемент после последнего скопированного элемента;
- итератор, возвращенный алгоритмом `copy_backward`, указывает на последний скопированный элемент, который является первым элементом в диапазоне.

```
19    // Копируем элементы массива a1 в results в обратном порядке
20    std::array<int, a1.size()> results{};
21    std::ranges::copy_backward(a1, results.end());
22    std::cout << "\n\nAfter copy_backward, results contains: ";
23    std::ranges::copy(results, output);
24
```

```
After copy_backward, results contains: 1 3 5 7 9
```

Алгоритмы move и move_backward

Вы можете использовать семантику перемещения с диапазонами. Алгоритмы `move` и `move_backward` библиотеки C++20 `std::ranges` (заголовок `<algorithm>`) работают аналогично алгоритмам `copy` и `copy_backward`, но не копируют, а перемещают элементы в указанный диапазон.

Алгоритм merge

Алгоритм C++20 `std::ranges::merge` (строка 27) объединяет два диапазона `input_range`, каждый из которых должен быть отсортирован в порядке возрастания. Запись результата происходит в контейнер, указанный итератором вывода в третьем аргументе. После этой операции `results2` содержит значения обоих диапазонов в отсортированном порядке. Вторая версия алгоритма `merge` использует пары итераторов, представляющие два диапазона. Обе версии позволяют вам использовать бинарную предикатную функцию, определяющую порядок сортировки путем сравнения двух аргументов и возвращающую `true`, если при сортировке первый аргумент следует считать меньшим, чем второй.

```
25    // Объединяем элементы a1 и a2 в results2 в отсортированном порядке
26    std::array<int, a1.size() + a2.size()> results2{};
27    std::ranges::merge(a1, a2, results2.begin());
28
29    std::cout << "\n\nAfter merge of a1 and a2, results2 contains: ";
30    std::ranges::copy(results2, output);
31
```

```
After merge of a1 and a2, results2 contains: 1 2 3 4 5 5 7 7 9 9
```

Алгоритм unique

Алгоритм C++20 `std::ranges::unique` (строка 34) находит уникальные значения в отсортированном диапазоне `forward_range`. После применения алгоритма `unique` к отсортированному диапазону с повторяющимися значениями в нем остается только одна копия каждого значения. Алгоритм возвращает поддиапазон, начинающийся с элемента, следующего за последним уникальным элементом, до конца исходного диапазона. Значения всех элементов в контейнере после последнего уникального значения становятся недоступными. Они не должны использоваться, и это еще один случай, когда вы можете уничтожить неиспользуемые элементы (строка 35). Вы также можете использовать бинарную предикатную функцию, указывающую способ сравнения двух элементов на предмет равенства.

```
32    // Удаляем дубликаты из вектора v
33    std::vector v(results2.begin(), results2.end());
34    auto [first, last]{std::ranges::unique(v)};
35    v.erase(first, last); // Удаляем элементы, которые не содержат значений
36
37    std::cout << "\n\nAfter unique v contains: ";
38    std::ranges::copy(v, output);
39
```

```
After unique, v contains: 1 2 3 4 5 7 9
```

Алгоритм reverse

Алгоритм C++20 `std::ranges::reverse` (строка 41) расставляет в обратном порядке элементы в диапазоне `bidirectional_range`, поддерживающем двунаправленные итераторы.

```
40    std::cout << "\n\nAfter reverse, a1 contains: ";
41    std::ranges::reverse(a1); // Переставляем элементы a1 в обратном порядке
42    std::ranges::copy(a1, output);
43
```

```
After reverse, a1 contains: 9 7 5 3 1
```

Алгоритм copy_if

В C++11 были добавлены алгоритмы `copy_if` и `copy_n`, а в C++20 — их версии для диапазонов. Алгоритм C++20 `std::ranges::copy_if` (строки 47–48) принимает в качестве аргументов диапазон `input_range`, итератор вывода и унарную предикатную функцию. Этот алгоритм вызывает унарную предикатную функцию для каждого элемента из диапазона `input_range` и копирует только те элементы, для которых эта функция возвращает `true`. Итератор вывода указывает на позицию для вывода элементов. В данном случае мы используем `back_inserter` для добавления элементов в вектор. Алгоритм возвращает `std::ranges::in_out_result`, содержащий два итератора:

- первый — итератор ввода, указывающий на конец диапазона `input_range`;
- второй — итератор вывода, указывающий на следующий элемент после последнего скопированного элемента в конечном контейнере.

```
44    // Копируем нечетные элементы массива a2 в вектор v2
45    std::vector<int> v2{};
46    std::cout << "\n\nAfter copy_if, v2 contains: ";
47    std::ranges::copy_if(a2, std::back_inserter(v2),
48       [](auto x){return x % 2 == 0;});
49    std::ranges::copy(v2, output);
50
```

```
After copy_if, v2 contains: 2 4
```

Алгоритм copy_n

Алгоритм C++20 `std::ranges::copy_n` (строка 54) копирует из позиции, указанной итератором ввода в первом аргументе (`a2.begin()`), определенное количество элементов, указанное во втором аргументе (`3`). Элементы копируются в позицию, указанную итератором вывода в третьем аргументе. В данном случае мы используем `back_inserter` для добавления элементов в вектор.

```
51    // Копируем три элемента массива a2 в вектор v3
52    std::vector<int> v3{};
53    std::cout << "\n\nAfter copy_n, v3 contains: ";
54    std::ranges::copy_n(a2.begin(), 3, std::back_inserter(v3));
55    std::ranges::copy(v3, output);
56    std::cout << "\n";
57 }
```

```
After copy_n, v3 contains: 2 4 5
```

14.4.9. Алгоритмы inplace_merge, unique_copy и reverse_copy

На илл. 14.10 демонстрируется работа алгоритмов `inplace_merge`, `unique_copy` и `reverse_copy` из пространства имен `std::ranges` C++20.

```
 1 // fig14_10.cpp
 2 // Алгоритмы inplace_merge, unique_copy и reverse_copy
 3 #include <algorithm>
 4 #include <array>
 5 #include <iostream>
 6 #include <iterator>
 7 #include <vector>
 8
 9 int main() {
10    std::array a1{1, 3, 5, 7, 9, 1, 3, 5, 7, 9};
11    std::ostream_iterator<int> output{std::cout, " "};
12
```

Окончание ⇨

```
13    std::cout << "array a1 contains: ";
14    std::ranges::copy(a1, output);
15
```

```
array a1 contains: 1 3 5 7 9 1 3 5 7 9
```

Илл. 14.10 | Алгоритмы inplace_merge, unique_copy и reverse_copy

Алгоритм inplace_merge

В строке 18 алгоритм C++20 `std::ranges::inplace_merge` объединяет два отсортированных диапазона `bidirectional_range`. В этом примере элементы диапазона `a1` (первый аргумент) от позиции `a1.begin()` до позиции (но не включая ее) `a1.begin() + 5` (второй аргумент) объединяются с элементами от позиции `a1.begin() + 5` (второй аргумент) до конца диапазона. Вы также можете передать алгоритму бинарную предикатную функцию, сравнивающую элементы в двух поддиапазонах и возвращающую `true`, если первый элемент следует считать меньшим, чем второй.

```
16    // Объединяем первую половину массива a1 со второй половиной
17    // таким образом, чтобы получить отсортированный массив
18    std::ranges::inplace_merge(a1, a1.begin() + 5);
19    std::cout << "\nAfter inplace_merge, a1 contains: ";
20    std::ranges::copy(a1, output);
21
```

```
After inplace_merge, a1 contains: 1 1 3 3 5 5 7 7 9 9
```

Алгоритм unique_copy

В строке 24 алгоритм C++20 `std::ranges::unique_copy` копирует уникальные элементы из отсортированного диапазона `input_range`, указанного в первом аргументе. Итератор вывода во втором аргументе указывает, куда надо копировать элементы. В данном случае `back_inserter` добавляет новые элементы в вектор `results1`, который увеличивает размер по мере необходимости. Вы также можете передать алгоритму бинарную предикатную функцию, указывающую, как проверять элементы на предмет равенства.

```
22    // Копируем только уникальные элементы массива a1 в results1
23    std::vector<int> results1{};
24    std::ranges::unique_copy(a1, std::back_inserter(results1));
25    std::cout << "\nAfter unique_copy, results1 contains: ";
26    std::ranges::copy(results1, output);
27
```

```
After unique_copy results1 contains: 1 3 5 7 9
```

Алгоритм reverse_copy

В строке 30 алгоритм C++20 `std::ranges::reverse_copy` копирует указанный в первом аргументе диапазон `bidirectional_range`, размещая его элементы в обратном порядке. Итератор вывода во втором аргументе указывает на место для

скопированных элементов. В данном случае back_inserter добавляет новые элементы в вектор results2, увеличивающий размер по мере необходимости.

```
28    // Копируем элементы массива a1 в results2 в обратном порядке
29    std::vector<int> results2{};
30    std::ranges::reverse_copy(a1, std::back_inserter(results2));
31    std::cout << "\nAfter reverse_copy, results2 contains: ";
32    std::ranges::copy(results2, output);
33    std::cout << "\n";
34 }
```

```
After reverse_copy, results2 contains: 9 9 7 7 5 5 3 3 1 1
```

14.4.10. Операции со множествами

На илл. 14.11 демонстрируется работа алгоритмов из пространства имен std::ranges C++20, предназначенных для операций с множествами: includes, set_difference, set_intersection, set_symmetric_difference и set_union. В дополнение к показанным в этом примере возможностям вы можете настраивать сравнение элементов в каждом алгоритме, передавая ему бинарную предикатную функцию, принимающую два аргумента и возвращающую true, если первый аргумент следует считать меньшим, чем второй.

```
 1 // fig14_11.cpp
 2 // Алгоритмы includes, set_difference, set_intersection,
 3 // set_symmetric_difference и set_union
 4 #include <array>
 5 #include <algorithm>
 6 #include <fmt/format.h> // Или в C++20: #include <format>
 7 #include <iostream>
 8 #include <iterator>
 9 #include <vector>
10
11 int main() {
12    std::array a1{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
13    std::array a2{4, 5, 6, 7, 8};
14    std::array a3{4, 5, 6, 11, 15};
15    std::ostream_iterator<int> output{std::cout, " "};
16
17    std::cout << "a1 contains: ";
18    std::ranges::copy(a1, output); // Выводим массив a1
19    std::cout << "\na2 contains: ";
20    std::ranges::copy(a2, output); // Выводим массив a2
21    std::cout << "\na3 contains: ";
22    std::ranges::copy(a3, output); // Выводим массив a3
23
```

```
a1 contains: 1 2 3 4 5 6 7 8 9 10
a2 contains: 4 5 6 7 8
a3 contains: 4 5 6 11 15
```

Илл. 14.11 | Алгоритмы includes, set_difference, set_intersection, set_symmetric_difference и set_union

Алгоритм includes

В строках 26 и 30 алгоритм C++20 `std::ranges::includes` сравнивает два отсортированных диапазона `input_range` и определяет, есть ли каждый элемент второго диапазона в первом диапазоне. Оба диапазона должны быть отсортированы одной и той же функцией. Алгоритм возвращает `true`, если все элементы второго диапазона находятся в первом диапазоне; в противном случае он возвращает `false`. В строке 26 алгоритм возвращает `true`, так как все элементы массива `a2` есть в массиве `a1`. В строке 30 алгоритм возвращает `false`, так как не все элементы массива `a3` есть в массиве `a1`.

```
24    // Выясняем, все ли элементы массива a2 есть в массиве a1
25    std::cout << fmt::format("\n\na1 {} a2",
26       std::ranges::includes(a1, a2) ? "includes" : "does not include");
27
28    // Выясняем, все ли элементы массива a3 есть в массиве a1
29    std::cout << fmt::format("\n\na1 {} a3",
30       std::ranges::includes(a1, a3) ? "includes" : "does not include");
31
```

```
a1 includes a2
a1 does not include a3
```

Алгоритм set_difference

В строке 34 алгоритм C++20 `std::ranges::set_difference` ищет те элементы первого отсортированного диапазона `input_range`, которых нет во втором отсортированном диапазоне `input_range`. Оба диапазона должны быть отсортированы одной и той же функцией. Элементы, которые отличаются, копируются в позицию, указанную итератором вывода в третьем аргументе. В данном случае мы добавляем их в вектор `difference` при помощи `back_inserter`.

```
32    // Ищем элементы массива a1, которых нет в массиве a2
33    std::vector<int> difference{};
34    std::ranges::set_difference(a1, a2, std::back_inserter(difference));
35    std::cout << "\n\nset_difference of a1 and a2 is: ";
36    std::ranges::copy(difference, output);
37
```

```
set_difference of a1 and a2 is: 1 2 3 9 10
```

Алгоритм set_intersection

В строках 40–41 алгоритм C++20 `std::ranges::set_intersection` ищет те элементы первого отсортированного диапазона `input_range`, которые есть во втором отсортированном диапазоне `input_range`. Оба диапазона должны быть отсортированы одной и той же функцией. Элементы, общие для обоих диапазонов, копируются в позицию, указанную итератором вывода в третьем аргументе. В данном случае мы добавляем их в вектор `intersection` при помощи `back_inserter`.

```
38    // Ищем элементы, которые есть в массиве a1 и в массиве a2
39    std::vector<int> intersection{};
40    std::ranges::set_intersection(a1, a2,
41       std::back_inserter(intersection));
42    std::cout << "\n\nset_intersection of a1 and a2 is: ";
43    std::ranges::copy(intersection, output);
44
```

```
set_intersection of a1 and a2 is: 4 5 6 7 8
```

Алгоритм set_symmetric_difference

В строках 48–49 алгоритм C++20 `std::ranges::set_symmetric_difference` ищет те элементы первого отсортированного диапазона `input_range`, которых нет во втором отсортированном диапазоне `input_range`, а также элементы второго диапазона, которых нет в первом. Оба диапазона должны быть отсортированы одной и той же функцией. Элементы обоих диапазонов `input_range`, соответствующие этому условию, копируются в позицию, указанную итератором вывода в третьем аргументе. В данном случае мы добавляем их в вектор `symmetricDifference` при помощи `back_inserter`.

```
45    // Ищем элементы массива a1, которых нет в массиве a3,
46    // и элементы массива a3, которых нет в массиве a1
47    std::vector<int> symmetricDifference{};
48    std::ranges::set_symmetric_difference(a1, a3,
49       std::back_inserter(symmetricDifference));
50    std::cout << "\n\nset_symmetric_difference of a1 and a3 is: ";
51    std::ranges::copy(symmetricDifference, output);
52
```

```
set_symmetric_difference of a1 and a3 is: 1 2 3 7 8 9 10 11 15
```

Алгоритм set_union

В строке 55 алгоритм C++20 `std::ranges::set_union` создает набор элементов, которые есть хотя бы в одном из двух отсортированных диапазонов `input_range` (или в обоих). Оба диапазона должны быть отсортированы одной и той же функцией. Элементы копируются в позицию, указанную итератором вывода в третьем аргументе. В данном случае мы добавляем их в вектор `unionSet` при помощи `back_inserter`. Элементы, которые есть в обоих диапазонах, копируются только из первого.

```
53    // Копируем все элементы, которые есть хотя бы в одном из двух множеств
54    std::vector<int> unionSet{};
55    std::ranges::set_union(a1, a3, std::back_inserter(unionSet));
56    std::cout << "\n\nset_union of a1 and a3 is: ";
57    std::ranges::copy(unionSet, output);
58    std::cout << "\n";
59 }
```

```
set_union of a1 and a3 is: 1 2 3 4 5 6 7 8 9 10 11 15
```

14.4.11. Алгоритмы lower_bound, upper_bound и equal_range

На илл. 14.12 демонстрируется работа алгоритмов `lower_bound`, `upper_bound` и `equal_range` из пространства имен `std::ranges` C++20[1]. В дополнение к показанным в этом примере возможностям вы можете настраивать сравнение элементов в каждом алгоритме, передавая ему бинарную предикатную функцию, принимающую два аргумента и возвращающую `true`, если первый аргумент следует считать меньшим, чем второй.

```
 1 // fig14_12.cpp
 2 // Алгоритмы lower_bound, upper_bound и
 3 // equal_range для отсортированных последовательностей
 4 #include <algorithm>
 5 #include <array>
 6 #include <iostream>
 7 #include <iterator>
 8
 9 int main() {
10    std::array values{2, 2, 4, 4, 4, 6, 6, 6, 6, 8};
11    std::ostream_iterator<int> output{std::cout, " "};
12
13    std::cout << "values contains: ";
14    std::ranges::copy(values, output);
15
```

```
values contains: 2 2 4 4 4 6 6 6 6 8
```

Илл. 14.12 | Алгоритмы lower_bound, upper_bound и equal_range для отсортированных последовательностей

Алгоритм lower_bound

В строке 17 алгоритм C++20 `std::ranges::lower_bound` ищет в отсортированном диапазоне `forward_range` первую позицию, в которую можно вставить второй аргумент, сохраняя сортировку диапазона в порядке возрастания. Алгоритм возвращает итератор, указывающий на найденную позицию.

```
16    // Ищем первую позицию для корректной вставки в values числа 6
17    auto lower{std::ranges::lower_bound(values, 6)};
18    std::cout << "\n\nLower bound of 6 is index: "
19       << (lower - values.begin());
20
```

```
Lower bound of 6 is index: 5
```

[1] В январе 2022 г. алгоритм std::ranges::equal_range не компилировался в последней версии компилятора clang++.

Алгоритм upper_bound

В строке 22 алгоритм C++20 `std::ranges::upper_bound` ищет в отсортированном диапазоне `forward_range` последнюю позицию, в которую можно вставить второй аргумент, сохраняя сортировку диапазона в порядке возрастания. Алгоритм возвращает итератор, указывающий на найденную позицию.

```
21    // Ищем последнюю позицию для корректной вставки в values числа 6
22    auto upper{std::ranges::upper_bound(values, 6)};
23    std::cout << "\nUpper bound of 6 is index: "
24       << (upper - values.begin());
25
```

```
Upper bound of 6 is index: 9
```

Алгоритм equal_range

Алгоритм C++20 `std::ranges::equal_range` (строка 27) выполняет операции алгоритмов `lower_bound` и `upper_bound`, возвращая их результаты в виде объекта `std::ranges::subrange`, который мы распаковываем в переменные `first` и `last`.

```
26    // Используем equal_range, чтобы найти первую и последнюю позицию
         для корректной вставки числа 6
27    auto [first, last]{std::ranges::equal_range(values, 6)};
28    std::cout << "\nUsing equal_range:\n Lower bound of 6 is index: "
29       << (first - values.begin());
30    std::cout << "\n Upper bound of 6 is index: "
31       << (last - values.begin());
32
```

```
Using equal_range:
Lower bound of 6 is index: 5
Upper bound of 6 is index: 9
```

Поиск позиций для вставки в отсортированных последовательностях

Алгоритмы `lower_bound`, `upper_bound` и `equal_range` часто используются для того, чтобы определить позицию вставки нового значения в отсортированную последовательность. В строке 36 алгоритм `lower_bound` ищет первую позицию, в которую можно вставить число 3, не нарушая порядок сортировки. В строке 43 алгоритм `upper_bound` ищет последнюю позицию, в которую можно вставить число 7, не нарушая порядок сортировки.

```
33    // Ищем первую позицию для корректной вставки в values числа 3
34    std::cout << "\n\nUse lower_bound to locate the first point "
35       << "at which 3 can be inserted in order";
36    lower = std::ranges::lower_bound(values, 3);
37    std::cout << "\n Lower bound of 3 is index: "
38       << (lower - values.begin());
39
40    // Ищем последнюю позицию для корректной вставки в values числа 7
```

```
41    std::cout << "\n\nUse upper_bound to locate the last point "
42       << "at which 7 can be inserted in order";
43    upper = std::ranges::upper_bound(values, 7);
44    std::cout << "\n Upper bound of 7 is index: "
45       << (upper - values.begin()) << "\n";
46 }
```

```
Use lower_bound to locate the first point at which 3 can be inserted in order
Lower bound of 3 is index: 2

Use upper_bound to locate the last point at which 7 can be inserted in order
Upper bound of 7 is index: 9
```

14.4.12. Алгоритмы min, max и minmax

На илл. 14.13 показана работа алгоритмов `min`, `max` и `minmax` из пространства имен `std`, а также перегруженной версии алгоритма `minmax` из пространства имен `std::ranges` (C++20). В отличие от алгоритмов, рассмотренных в разделе 14.4.5 и работающих с диапазонами, алгоритмы `min`, `max` и `minmax` из пространства имен `std` работают с двумя значениями, переданными в аргументах. Алгоритм `std::ranges::minmax` возвращает минимальное и максимальное значения диапазона.

Алгоритмы min и max с двумя параметрами

Каждый из алгоритмов `min` и `max` (строки 8–12) получает по два аргумента и возвращает минимальное или максимальное значение соответственно. Алгоритмы также имеют перегруженные версии, принимающие в качестве третьего аргумента бинарную предикатную функцию для пользовательского сравнения, которая определяет, следует ли считать первый аргумент меньшим, чем второй.

```
 1 // fig14_13.cpp
 2 // Алгоритмы min, max и minmax
 3 #include <array>
 4 #include <algorithm>
 5 #include <iostream>
 6
 7 int main() {
 8    std::cout << "Minimum of 12 and 7 is: " << std::min(12, 7)
 9       << "\nMaximum of 12 and 7 is: " << std::max(12, 7)
10       << "\nMinimum of 'G' and 'Z' is: '" << std::min('G', 'Z') << "'"
11       << "\nMaximum of 'G' and 'Z' is: '" << std::max('G', 'Z') << "'"
12       << "\nMinimum of 'z' and 'Z' is: '" << std::min('z', 'Z') << "'";
13
```

```
Minimum of 12 and 7 is: 7
Maximum of 12 and 7 is: 12
Minimum of 'G' and 'Z' is: 'G'
Maximum of 'G' and 'Z' is: 'Z'
Minimum of 'z' and 'Z' is: 'Z'
```

Илл. 14.13 | Алгоритмы min, max и minmax

Алгоритм minmax с двумя аргументами (C++11)

В C++11 был добавлен алгоритм `minmax` с двумя аргументами (строка 15), возвращающий пару значений, содержащих наименьший и наибольший элементы соответственно. Здесь мы использовали структурное связывание для распаковки значений в `smaller` и `larger`. Вторая версия алгоритма `minmax` принимает в качестве третьего аргумента бинарную предикатную функцию для пользовательского сравнения, которая определяет, следует ли считать первый аргумент меньшим, чем второй.

```
14    // Выясняем, какой из двух аргументов меньше, а какой больше
15    auto [smaller, larger]{std::minmax(12, 7)};
16    std::cout << "\n\nMinimum of 12 and 7 is: " << smaller
17       << "\nMaximum of 12 and 7 is: " << larger;
18
```

```
Minimum of 12 and 7 is: 7
Maximum of 12 and 7 is: 12
```

Алгоритм minmax библиотеки ranges (C++20)

Добавленный в C++20 алгоритм `std::ranges::minmax` (строка 25) возвращает пару значений, содержащих минимальный и максимальный элементы в диапазоне `input_range` или в `initializer_list`. Мы снова используем структурное связывание для распаковки найденных значений в `smallest` и `largest` соответственно. Вы также можете передать в качестве второго аргумента бинарную предикатную функцию, сравнивающую элементы и определяющую, следует ли считать первый элемент меньшим, чем второй.

```
19    std::array items{3, 100, 52, 77, 22, 31, 1, 98, 13, 40};
20    std::ostream_iterator<int> output{std::cout, " "};
21
22    std::cout << "\n\nitems: ";
23    std::ranges::copy(items, output);
24
25    auto [smallest, largest]{std::ranges::minmax(items)};
26    std::cout << "\nMinimum value in items: " << smallest
27       << "\nMaximum value in items is: " << largest << "\n";
28 }
```

```
items: 3 100 52 77 22 31 1 98 13 40
Minimum value in items: 1
Maximum value in items is: 100
```

14.4.13. Алгоритмы gcd, lcm, iota, reduce и partial_sum (заголовок <numeric>)

В разделе 8.19.2 мы использовали алгоритм `accumulate` из заголовка `<numeric>`. На илл. 14.14 показана работа других алгоритмов из заголовка `<numeric>`: `gcd`,

`lcm`, `iota`, `reduce` и `partial_sum`. Алгоритмы этого заголовка работают с обычными диапазонами и, как ожидается, будут перегружены для работы с диапазонами в C++23[1].

Алгоритм gcd

Алгоритм `gcd` (строки 14 и 15) принимает два целочисленных аргумента и возвращает их наибольший общий делитель.

```
 1 // fig14_14.cpp
 2 // Алгоритмы gcd, lcm, iota, reduce и partial_sum
 3 #include <array>
 4 #include <algorithm>
 5 #include <functional>
 6 #include <iostream>
 7 #include <iterator>
 8 #include <numeric>
 9
10 int main() {
11    std::ostream_iterator<int> output{std::cout, " "};
12
13    // Вычисляем наибольший общий делитель двух целых чисел
14    std::cout << "std::gcd(75, 20): " << std::gcd(75, 20)
15       << "\nstd::gcd(17, 13): " << std::gcd(75, 13);
16
```

```
std::gcd(75, 20): 5
std::gcd(17, 13): 1
```

Илл. 14.14 | Алгоритмы gcd, lcm, iota, reduce и partial_sum

Алгоритм lcm

Алгоритм `lcm` (строки 18 и 19) принимает два целочисленных аргумента и возвращает их наименьшее общее кратное.

```
17    // Вычисляем наименьшее общее кратное двух целых чисел
18    std::cout << "\n\nstd::lcm(3, 5): " << std::lcm(3, 5)
19       << "\nstd::lcm(12, 9): " << std::lcm(12, 9);
20
```

```
std::lcm(3, 5): 15
std::lcm(12, 9): 36
```

Алгоритм iota

Алгоритм `iota` (строка 23) заполняет обычный диапазон последовательностью значений, начинающихся со значения в третьем аргументе. Первые два аргумента

[1] Барри Ревзин (Barry Revzin), Конор Хукстра (Conor Hoekstra) и Тим Сонг (Tim Song), «A Plan for C++23 Ranges», 14 октября 2020 г. Был доступен: 15 сентября 2023 г. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2214r0.html#algorithms.

должны быть однонаправленными итераторами, представляющими обычный диапазон для заполнения. Тип последнего аргумента должен поддерживать оператор ++.

```
21    // Заполняем массив целыми числами, используя алгоритм std::iota
22    std::array<int, 5> ints{};
23    std::iota(ints.begin(), ints.end(), 1);
24    std::cout << "\n\nints: ";
25    std::ranges::copy(ints, output);
26
```

```
ints: 1 2 3 4 5
```

Алгоритм reduce

Алгоритм `reduce` (строки 29 и 31) выполняет свертку значений всех элементов диапазона в одно значение. Первый и второй аргументы должны быть итераторами ввода. Вызов в строке 29 неявно добавляет элементы обычного диапазона. Вызов в строке 31 предоставляет пользовательское значение инициализатора (`1`) и бинарную функцию, определяющую способ выполнения свертки. Здесь мы используем `std::multiplies{}` (заголовок `<functional>`) — предопределенный функциональный объект[1], перемножающий два своих аргумента и возвращающий результат. Фигурные скобки (`{}`) создают временный объект `std::multiplies` и вызывают его конструктор. Каждый функциональный объект имеет перегруженную функцию `operator()`. Внутри алгоритма `reduce` функция `operator()` вызывается для получения результата. Любая коммутативная и ассоциативная бинарная функция, принимающая два значения одного и того же типа и возвращающая результат этого типа, может быть передана в качестве четвертого аргумента. В разделе 14.5 вы увидите, что заголовок `<functional>` определяет бинарные функциональные объекты для сложения, вычитания, умножения, деления, вычисления по модулю и других операций.

```
27    // Выполняем свертку элементов контейнера в одно значение
28    std::cout << "\n\nsum of ints: "
29       << std::reduce(ints.begin(), ints.end())
30       << "\nproduct of ints: "
31       << std::reduce(ints.begin(), ints.end(), 1, std::multiplies{});
32
```

```
sum of ints: 15
product of ints: 120
```

Различия между алгоритмами reduce и accumulate

Алгоритм `reduce` похож на алгоритм `accumulate`, но `reduce` не гарантирует порядок, в котором обрабатываются элементы. В главе 17 вы увидите, что из-за этого

[1] Подробнее о предопределенных функциональных объектах — в разделе 14.5.

различия алгоритм `reduce` может быть распараллелен для ускорения работы, в отличие от `accumulate`[1].

Алгоритм partial_sum

Алгоритм `partial_sum` (строки 37 и 39) вычисляет сумму всех элементов обычного диапазона от начала диапазона до текущего элемента включительно. По умолчанию эта версия алгоритма `partial_sum` использует функциональный объект `std::plus`, возвращающий сумму двух аргументов. Для массива `ints`, заполненного в строке 23 числами `1`, `2`, `3`, `4` и `5`, вызов алгоритма в строке 37 выводит следующие результаты:

- `1` (первый элемент массива `ints`);
- `3` (1 + 2);
- `6` (1 + 2 + 3);
- `10` (1 + 2 + 3 + 4);
- `15` (1 + 2 + 3 + 4 + 5).

```
33    // Вычисляем частичные суммы элементов массива ints
34    std::cout << "\n\nints: ";
35    std::ranges::copy(ints, output);
36    std::cout << "\n\npartial_sum of ints using std::plus by default: ";
37    std::partial_sum(ints.begin(), ints.end(), output);
38    std::cout << "\npartial_sum of ints using std::multiplies: ";
39    std::partial_sum(ints.begin(), ints.end(), output, std::multiplies{});
40    std::cout << "\n";
41 }
```

```
ints: 1 2 3 4 5

partial_sum of ints using std::plus by default: 1 3 6 10 15
partial_sum of ints using std::multiplies: 1 2 6 24 120
```

В строке 39 вызывается перегруженная версия алгоритма `partial_sum`, получающая бинарную функцию, определяющую способ выполнения вычислений. В этом случае мы использовали предопределенный бинарный функциональный объект `std::multiplies{}`, получая вместо сумм произведения значений от начала контейнера до текущего элемента:

- `1` (первый элемент массива `ints`);
- `2` (1 * 2);
- `6` (1 * 2 * 3);
- `24` (1 * 2 * 3 * 4);
- `120` (1 * 2 * 3 * 4 * 5).

[1] Сай Бренд (Sy Brand), «std::accumulate vs. std::reduce», 15 мая 2018 г. Был доступен: 15 сентября 2023 г. https://blog.tartanllama.xyz/accumulate-vs-reduce/.

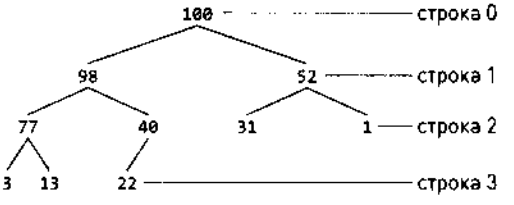
Этот вызов алгоритма `partial_sum` фактически вычисляет факториалы чисел от 1 до 5 (то есть `1!`, `2!`, `3!`, `4!` и `5!`).

14.4.14. Пирамидальная сортировка кучи и приоритетные очереди

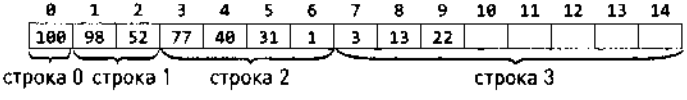
В разделе 13.10.3 был рассмотрен адаптер контейнера `priority_queue` (приоритетная очередь). Способ хранения элементов, добавленных в `priority_queue`, позволяет удалять их в порядке приоритета. Элемент с высшим приоритетом всегда удаляется первым: как правило, элемент с высшим приоритетом имеет наибольшее значение, но это можно настраивать. Поиск элемента с высшим приоритетом эффективен, если размещать элементы в структуре данных, которая называется **кучей (heap)**, — не путайте с той кучей, которую C++ использует для динамического выделения памяти. Обычно приоритетные очереди используются в планировании процессов операционной системы.

Структура данных кучи

Куча обычно реализуется в виде бинарного дерева. **Макс-куча (max heap)**, также называемая убывающей кучей, хранит свое максимальное значение «сверху», то есть в корневом узле, а значение любого дочернего узла меньше или равно значению родительского узла. **Мин-куча (min heap)**, также называемая возрастающей кучей, хранит свое минимальное значение «сверху», то есть в корневом узле, а значение любого дочернего узла больше или равно значению родительского узла. Кучи могут содержать повторяющиеся значения. На диаграмме показано бинарное дерево, представляющее макс-кучу:



Как правило, кучи хранятся в массивах и подобных им структурах данных, таких как `array`, `vector` или `deque`; все они используют итераторы произвольного доступа. Показанная на предыдущей диаграмме куча размещается в массиве `array` таким образом:



Вы можете убедиться, что этот массив представляет макс-кучу. Для любого индекса `n` в массиве вы можете найти индекс родительского узла, выполнив целочисленную арифметическую операцию:

```
(n - 1) / 2
```

Затем вы можете проверить, что значение дочернего узла меньше или равно значению родительского узла. Наибольшее значение макс-кучи всегда находится в корне бинарного дерева, что соответствует элементу массива с индексом `0`.

Алгоритмы, работающие с кучей

На илл. 14.15 показана работа четырех алгоритмов из пространства имен `std::ranges` (C++20), обрабатывающих кучи. Сначала мы рассмотрим алгоритмы `make_heap` и `sort_heap`, реализующие два шага в алгоритме пирамидальной сортировки, который в худшем случае имеет время выполнения $O(n \log n)$[1]:

- Шаг 1: упорядочить элементы контейнера в виде кучи.
- Шаг 2: удалить элементы из кучи таким образом, чтобы получить отсортированную последовательность.

Затем мы рассмотрим алгоритмы `push_heap` и `pop_heap`. Адаптер контейнера `priority_queue`[2] использует их «под капотом», чтобы поддерживать кучу по мере добавления и удаления элементов.

```
 1 // fig14_15.cpp
 2 // Алгоритмы make_heap, sort_heap, push_heap и pop_heap
 3 #include <iostream>
 4 #include <algorithm>
 5 #include <array>
 6 #include <vector>
 7 #include <iterator>
 8
 9 int main() {
10    std::ostream_iterator<int> output{std::cout, " "};
11
```

Илл. 14.15 | Алгоритмы make_heap, sort_heap, push_heap и pop_heap

Инициализация и отображение heapArray

В строке 12 создается и инициализируется массив `heapArray`, содержащий 10 целых чисел в неотсортированном порядке. В строке 14 массив выводится на экран перед преобразованием в кучу и сортировкой элементов.

---

[1] «Heapsort». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Heapsort.

[2] Вы можете ознакомиться с реализацией priority_queue с использованием push_heap и pop_heap в стандартной библиотеке Microsoft C++ с открытым исходным кодом: https://github.com/microsoft/STL/blob/main/stl/inc/queue. Был доступен: 15 сентября 2023 г.

```
12    std::array heapArray{3, 100, 52, 77, 22, 31, 1, 98, 13, 40};
13    std::cout << "heapArray before make_heap:\n";
14    std::ranges::copy(heapArray, output);
15
```

```
heapArray before make_heap:
3 100 52 77 22 31 1 98 13 40
```

Алгоритм make_heap

В строке 16 алгоритм C++20 `std::ranges::make_heap` структурирует элементы диапазона `random_access_range` в виде кучи, которую затем можно будет обработать алгоритмом `sort_heap` для создания отсортированной последовательности. В строке 18 элементы массива `heapArray`, структурированные в виде кучи, выводятся на экран. Диапазон `random_access_range` поддерживает итераторы произвольного доступа, поэтому алгоритм `make_heap` работает с контейнерами `array`, `vector` и `deque`.

```
16    std::ranges::make_heap(heapArray); // Создаем кучу из массива heapArray
17    std::cout << "\nheapArray after make_heap:\n";
18    std::ranges::copy(heapArray, output);
19
```

```
heapArray after make_heap:
100 98 52 77 40 31 1 3 13 22
```

Алгоритм sort_heap

В строке 20 алгоритм C++20 `std::ranges::sort_heap` сортирует элементы диапазона `random_access_range`. Диапазон уже должен быть кучей. В строке 22 отображается отсортированная куча.

```
20    std::ranges::sort_heap(heapArray); // Сортируем алгоритмом sort_heap
21    std::cout << "\nheapArray after sort_heap:\n";
22    std::ranges::copy(heapArray, output);
23
```

```
heapArray after sort_heap:
1 3 13 22 31 40 52 77 98 100
```

Алгоритмы push_heap и pop_heap для поддержки кучи

Теперь мы рассмотрим алгоритмы, которые контейнер `priority_queue` использует «под капотом» для добавления элемента в кучу и для удаления элемента из кучи. Обе операции имеют вычислительную сложность $O(\log n)$[1]. В строках 25–33 определяется лямбда `push`, добавляющая одно целочисленное значение в кучу, находящуюся в объекте `vector`. Для этого лямбда выполняет следующие задачи:

[1] «Binary Heap». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Binary_heap.

- В строке 28 добавляется одно целочисленное значение в конец вектора.
- В строке 29 вызывается алгоритм C++20 `std::ranges::push_heap`, который принимает последний элемент аргумента `random_access_range` (`heap`) и добавляет его в кучу. Каждый раз при вызове алгоритма `push_heap` предполагается, что в кучу добавляется последний элемент, а остальные элементы уже расположены в виде кучи. Если элемент, добавленный в строке 28, является единственным элементом диапазона, диапазон уже является кучей. В противном случае алгоритм `push_heap` заново структурирует элементы в виде кучи.
- В строке 31 отображается структура данных обновленной кучи после добавления каждого значения.

```
24    // Лямбда для добавления целочисленного значения в кучу
25    auto push{
26       [&](std::vector<int>& heap, int value) {
27          std::cout << "\n\npushing " << value << " onto heap";
28          heap.push_back(value); // Добавляем значение в кучу
29          std::ranges::push_heap(heap); // Добавляем последний элемент в кучу
30          std::cout << "\nheap: ";
31          std::ranges::copy(heap, output);
32       }
33    };
34
```

В строках 36–44 определяется лямбда `pop`, которая удаляет наибольшее значение из кучи. Для этого лямбда выполняет следующие задачи:

- В строке 38 вызывается алгоритм C++20 `std::ranges::pop_heap` для удаления наибольшего значения из кучи. Алгоритм предполагает, что диапазон `random_access_range` является кучей. Сначала алгоритм заменяет самый большой элемент кучи, расположенный в `heap.begin()`, на ее последний элемент, расположенный перед `heap.end()`. Затем он проверяет, что все элементы от начала до конца диапазона (кроме последнего элемента) по-прежнему представляют собой кучу. Алгоритм `pop_heap` не изменяет количество элементов в диапазоне.
- В строке 39 отображается значение последнего элемента вектора — это то значение, которое только что было удалено из кучи, но все еще остается в векторе.
- В строке 40 удаляется последний элемент вектора. В контейнере остаются только элементы, представляющие кучу.
- В строке 42 выводится текущее содержимое кучи.

```
35    // Лямбда для удаления элемента из кучи
36    auto pop{
37       [&](std::vector<int>& heap) {
38          std::ranges::pop_heap(heap); // Удаляем наибольший элемент из кучи
39          std::cout << "\n\npopping highest priority item: " << heap.back();
```

```
40          heap.pop_back(); // Удаляем последний элемент вектора
41          std::cout << "\nheap: ";
42          std::ranges::copy(heap, output);
43       }
44    };
45
```

Демонстрация работы с кучей

В строке 46 определяется пустой вектор `<int>`, в котором будет поддерживаться куча. В строках 49–51 вызывается лямбда `push`, чтобы добавить значения `3`, `52` и `100` в кучу. Обратите внимание, что при добавлении каждого значения наибольшее значение всегда сохраняется в первом элементе вектора, а сами элементы не хранятся в отсортированном порядке.

```
46    std::vector<int> heapVector{};
47
48    // Размещаем пять целых чисел в heapVector, сохраняя структуру кучи
49    for (auto value : {3, 52, 100}) {
50       push(heapVector, value);
51    }
52
```

```
pushing 3 onto heap
heap: 3

pushing 52 onto heap
heap: 52 3

pushing 100 onto heap
heap: 100 3 52
```

Далее в строке 53 вызывается лямбда `pop` для удаления из кучи элемента с высшим приоритетом (`100`). Обратите внимание, что наибольшее значение, оставшееся в куче после удаления элемента (`52`), теперь находится в первом элементе вектора. В строке 54 в кучу добавляется значение `22`.

```
53    pop(heapVector); // Удаляем наибольший элемент
54    push(heapVector, 22); // Добавляем элемент
55
```

```
popping highest priority item: 100
heap: 52 3

pushing 22 onto heap
heap: 52 3 22
```

Далее в строке 56 удаляется элемент с высшим приоритетом (`52`). Опять же, наибольшее значение (`22`) теперь находится в первом элементе вектора. В строке 57

в кучу добавляется значение 77. Теперь это самое большое значение, поэтому оно находится в первом элементе вектора.

```
56    pop(heapVector); // Удаляем наибольший элемент
57    push(heapVector, 77); // Добавляем элемент
58
```

```
popping highest priority item: 52
heap: 22 3

pushing 77 onto heap
heap: 77 3 22
```

Наконец, в строках 59–61 из кучи удаляются три оставшихся элемента. Обратите внимание, что после выполнения инструкции в строке 59 самый большой оставшийся элемент (22) становится первым элементом вектора.

```
59    pop(heapVector); // Удаляем наибольший элемент
60    pop(heapVector); // Удаляем наибольший элемент
61    pop(heapVector); // Удаляем наибольший элемент
62    std::cout << "\n";
63 }
```

```
popping highest priority item: 77
heap: 22 3

popping highest priority item: 22
heap: 3

popping highest priority item: 3
heap:
```

14.5. Функциональные объекты (функторы)

Как было показано, многие алгоритмы стандартной библиотеки позволяют передавать в алгоритм лямбда-выражение или указатель на функцию, чтобы уточнить способ выполнения задачи. Любой алгоритм, который может получать лямбда-выражение или указатель на функцию, может получать и **функциональный объект (function object)**, также называемый **функтором (functor)**. Функтор — это объект класса, перегружающего оператор вызова функции (оператор «скобки») функцией с именем `operator()`. Перегруженная функция `operator()` должна соответствовать требованиям алгоритма к количеству параметров и к типу возвращаемого значения.

Функциональные объекты могут синтаксически и семантически использоваться подобно лямбде или указателю на функцию. Функция `operator()` вызывается через имя объекта, за которым следуют круглые скобки, содержащие аргументы. Большинство алгоритмов могут взаимозаменяемо использовать и лямбды, и указатели на функции, и функциональные объекты.

Преимущества функциональных объектов

Функциональные объекты дают ряд преимуществ по сравнению с функциями или указателями на функции:

- Функциональный объект является объектом класса, поэтому он может содержать нестатические данные для поддержания состояния конкретного функционального объекта или статические данные для поддержания общего состояния всех функциональных объектов данного класса. Кроме того, тип класса функционального объекта может использоваться как тип аргумента по умолчанию для параметров-типов шаблона[1].
- Возможно, самое главное преимущество: компилятор может **встраивать (inline)** функциональные объекты для повышения быстродействия. Функция `operator()` функционального объекта обычно определяется в теле класса, что неявно определяет ее как встроенную, то есть `inline`-функцию. Когда функция `operator()` определена вне тела класса, она может быть явно объявлена как встроенная. Компиляторы реализуют лямбды как функциональные объекты, поэтому они тоже могут быть встроенными. С другой стороны, компиляторы обычно не встраивают функции, вызываемые через указатели: такие указатели могут указывать на любую функцию с соответствующими параметрами и типом возвращаемого значения, поэтому компилятор не знает, какую из функций надо встраивать[2].

Предопределенные функциональные объекты стандартной библиотеки

Заголовок `<functional>` содержит много предопределенных функциональных объектов. Каждый из них реализован как шаблон класса. В следующей таблице перечислены некоторые часто используемые функциональные объекты стандартной библиотеки. Каждый функциональный объект равенства или сравнения имеет соответствующую версию с тем же именем в пространстве имен `std::ranges` (C++20). Большинство функциональных объектов являются бинарными: они получают два аргумента и возвращают результат. Функциональные объекты `logical_not` и `negate` являются унарными: они получают один аргумент и возвращают результат.

Функциональный объект	Тип	Функциональный объект	Тип
`divides<T>`	арифметический	`logical_or<T>`	логический
`equal_to<T>`	реляционный	`minus<T>`	арифметический
`greater<T>`	реляционный	`modulus<T>`	арифметический

[1] «Function Objects in the C++ Standard Library», 15 марта 2019 г. Был доступен: 15 сентября 2023 г. https://docs.microsoft.com/en-us/cpp/standard-library/function-objects-in-the-stl.

[2] Скотт Майерс (Scott Meyers), «Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library», с. 201–202: Pearson Education, 2001.

Функциональный объект	Тип	Функциональный объект	Тип
`greater_equal<T>`	реляционный	`negate<T>`	арифметический
`less<T>`	реляционный	`not_equal_to<T>`	реляционный
`less_equal<T>`	реляционный	`plus<T>`	арифметический
`logical_and<T>`	логический	`multiplies<T>`	арифметический
`logical_not<T>`	логический		

Рассмотрим функциональный объект для сравнения двух целых чисел:

```
std::less<int> smaller{};
```

Мы можем использовать имя объекта (`smaller`) для вызова функции `operator()` таким образом:

```
smaller(10, 7)
```

В этом случае `smaller` вернет `false`, потому что `10` не меньше 7. Такой алгоритм, как `sort`, мог бы использовать эту информацию для сортировки значений в порядке возрастания. Мы использовали функциональный объект `less<T>` в разделе 13.9, указывая порядок сортировки ключей в контейнерах `set` и `map`.

Вы можете ознакомиться с полным списком функциональных объектов по адресу

https://en.cppreference.com/w/cpp/utility/functional

и в разделе «Function Objects» стандарта C++[1]. Алгоритмы `std::ranges`, сравнивающие элементы для упорядочивания, используют `less<T>` в качестве предикатной функции по умолчанию. Напомним, что многие из перегруженных алгоритмов стандартной библиотеки, выполняющих сравнения, могут получать бинарную функцию, определяющую, является ли ее первый аргумент меньшим, чем второй, — это и есть задача функционального объекта `less<T>`.

Алгоритм accumulate

На илл. 14.16 демонстрируется алгоритм `std::accumulate` (заголовок `<numeric>`), вычисляющий сумму квадратов элементов массива. Алгоритмы из заголовка `<numeric>` не имеют перегруженных версий в пространстве имен `std::ranges`, поэтому они используют обычные диапазоны. Добавление перегруженных версий алгоритмов для диапазонов планируется в C++23[2]. Алгоритм `accumulate` имеет две перегруженные версии. Версия с тремя аргументами обрабатывает элементы обычного диапазона по умолчанию. Версия с четырьмя аргументами получает

[1] «General Utilities Library — Function Objects». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/function.objects.

[2] Кристофер Ди Белла (Christopher Di Bella), «A Concept Design for the Numeric Algorithms», 2 августа 2019 г. Был доступен: 15 сентября 2023 г. http://wg21.link/p1813r0.

в качестве последнего аргумента бинарную функцию, указывающую способ выполнения вычислений. Этот аргумент может быть представлен в виде:

- указателя на бинарную функцию, принимающую два параметра с типом элементов обычного диапазона и возвращающую результат этого же типа;
- бинарного функционального объекта, в котором функция `operator()` принимает два параметра с типом элементов обычного диапазона и возвращает результат этого же типа;
- лямбда-выражения, принимающего два параметра с типом элемента обычного диапазона и возвращающего результат этого же типа.

В следующем примере алгоритм `accumulate` вызывается с указателем на функцию, с функциональным объектом и с лямбда-выражением.

```
1 // fig14_16.cpp
2 // Функциональные объекты
3 #include <array>
4 #include <algorithm>
5 #include <functional>
6 #include <iostream>
7 #include <iterator>
8 #include <numeric>
9
```

Илл. 14.16 | Функциональные объекты

Функция sumSquares

В строках 12–14 определяется функция `sumSquares`, принимающая два аргумента одного типа и возвращающая значение этого же типа, — это соответствует требованиям к бинарным функциям, которые алгоритм `accumulate` может принимать в качестве аргумента. Функция `sumSquares` возвращает сумму своего первого аргумента `total` и квадрата второго аргумента `value`.

```
10 // Бинарная функция возвращает сумму первого аргумента total
11 // и квадрата второго аргумента value
12 int sumSquares(int total, int value) {
13    return total + value * value;
14 }
15
```

Класс SumSquaresClass

В строках 19–25 определяется класс `SumSquaresClass`[1]. Перегруженная функция `operator()` принимает два целочисленных параметра и возвращает целочисленное

[1] Этот класс обрабатывает только целые числа, но может быть реализован как шаблон класса, обрабатывающий много типов; мы будем определять пользовательские шаблоны классов в главе 15.

значение — это соответствует требованиям к бинарным функциям, которые можно вызывать при обработке обычного диапазона целых чисел. Функция `operator()` возвращает сумму первого аргумента `total` и квадрата второго аргумента `value`.

```
// Класс SumSquaresClass определяет перегруженную функцию operator(),
// которая возвращает сумму первого аргумента total
// и квадрата второго аргумента value
class SumSquaresClass {
public:
   // Прибавляем квадрат value к total и возвращаем результат
   int operator()(int total, int value) {
      return total + value * value;
   }
};

```

Вызовы алгоритма accumulate

Мы вызываем алгоритм `accumulate` три раза:

- В строках 36–37 алгоритм вызывается с указателем на функцию `sumSquares` в качестве последнего аргумента.
- В строках 44–45 алгоритм вызывается с функциональным объектом `SumSquaresClass` в качестве последнего аргумента. Аргумент `SumSquaresClass{}` в строке 45 создает временный объект `SumSquaresClass` и вызывает его конструктор. Затем этот функциональный объект передается в алгоритм `accumulate`, вызывающий функцию `operator()` объекта `SumSquaresClass`.
- В строках 50–51 алгоритм вызывается с лямбда-выражением, выполняющим те же задачи, что и функция `sumSquares` и перегруженная функция `operator()` класса `SumSquaresClass`.

```
int main() {
   std::array integers{1, 2, 3, 4};
   std::ostream_iterator<int> output{std::cout, " "};

   std::cout << "array integers contains: ";
   std::ranges::copy(integers, output);

   // Вычисляем сумму квадратов элементов массива integers
   // с помощью бинарной функции sumSquares
   int result{std::accumulate(integers.cbegin(), integers.cend(),
      0, sumSquares)};

   std::cout << "\n\nSum of squares\n"
      << "via binary function sumSquares: " << result;

   // Вычисляем сумму квадратов элементов массива integers
   // с помощью бинарного функционального объекта
   result = std::accumulate(integers.cbegin(), integers.cend(),
```

```
45       0, SumSquaresClass{});
46
47    std::cout << "\nvia a SumSquaresClass function object: " << result;
48
49    // Вычисляем сумму квадратов элементов массива integers
50    result = std::accumulate(integers.cbegin(), integers.cend(),
51       0, [](auto total, auto value){return total + value * value;});
52
53    std::cout << "\nvia a lambda: " << result << "\n";
54 }
```

```
array integers contains: 1 2 3 4

Sum of squares
via binary function sumSquares: 30
via a SumSquaresClass function object: 30
via a lambda: 30
```

Каждый вызов алгоритма `accumulate` использует функцию из аргумента следующим образом:

- При первом вызове алгоритм передает значение своего третьего аргумента (`0` в этом примере) и значение первого элемента массива `integers` (`1` в этом примере). Вычисляется и возвращается результат выражения `0 + 1 * 1`, что равно `1`.
- При втором вызове алгоритм передает предыдущий результат (`1`) и значение следующего элемента `integers` (`2`). Вычисляется и возвращается результат выражения `1 + 2 * 2`, что равно `5`.
- При третьем вызове алгоритм передает предыдущий результат (`5`) и значение следующего элемента массива `integers` (`3`). Вычисляется и возвращается результат выражения `5 + 3 * 3`, что равно `14`.
- При последнем вызове алгоритм передает предыдущий результат (`14`) и значение следующего элемента `integers` (`4`). Вычисляется и возвращается результат выражения `14 + 16`, что равно `30`.

На этом этапе алгоритм `accumulate` достигает конца обычного диапазона, указанного первыми двумя аргументами, и возвращает результат (`30`) последнего вызова функции из аргумента.

14.6. Проекции

При работе с объектами, содержащими несколько элементов данных, каждый алгоритм `std::ranges` (C++20) может использовать **проекцию (projection)**, чтобы выбрать для обработки лишь часть каждого объекта. Рассмотрим объекты класса `Employee`, каждый из которых содержит имя, фамилию и зарплату. Вместо того чтобы сортировать сотрудников по всем трем элементам данных, вы можете отсортировать их только по зарплате. На илл. 14.17 массив объектов `Employee` сортируется по зарплате: сначала в порядке возрастания, затем в порядке убывания. В строках 11–22

определяется класс Employee, а в строках 25–29 определяется перегруженная функция operator<<, позволяющая удобно выводить информацию на экран.

```
 1 // fig14_17.cpp
 2 // Проекции с алгоритмами std::ranges
 3 #include <array>
 4 #include <algorithm>
 5 #include <fmt/format.h>
 6 #include <iostream>
 7 #include <iterator>
 8 #include <string>
 9 #include <string_view>
10
11 class Employee {
12 public:
13    Employee(std::string_view first, std::string_view last, int salary)
14       : m_first{first}, m_last{last}, m_salary{salary} {}
15    std::string getFirst() const {return m_first;}
16    std::string getLast() const {return m_last;}
17    int getSalary() const {return m_salary;}
18 private:
19    std::string m_first;
20    std::string m_last;
21    int m_salary;
22 };
23
24 // operator<< для объектов Employee
25 std::ostream& operator<<(std::ostream& out, const Employee& e) {
26    out << fmt::format("{:10}{:10}{}",
27       e.getLast(), e.getFirst(), e.getSalary());
28    return out;
29 }
30
```

Илл. 14.17 | Проекции с алгоритмами std::ranges

Определение и отображение array<Employee>

В строках 32–36 определяется array<Employee> и инициализируется тремя объектами Employee. В строке 41 отображается информация о сотрудниках, поэтому позже мы сможем убедиться в правильной сортировке.

```
31 int main() {
32    std::array employees{
33       Employee{"Jason", "Red", 5000},
34       Employee{"Ashley", "Green", 7600},
35       Employee{"Matthew", "Indigo", 3587}
36    };
37
38    std::ostream_iterator<Employee> output{std::cout, "\n"};
39
40    std::cout << "Employees:\n";
```

```
41    std::ranges::copy(employees, output);
42
```

```
Employees:
Red       Jason     5000
Green     Ashley    7600
Indigo    Matthew   3587
```

Проекция для сортировки array<Employee> в порядке возрастания

В строках 45–46 вызывается алгоритм `std::ranges::sort` с тремя аргументами:

- Первый аргумент (`employees`) — диапазон для сортировки.
- Второй аргумент (`{}`) — бинарная предикатная функция, которую алгоритм использует для сравнения элементов при определении порядка сортировки. Обозначение `{}` указывает, что алгоритм должен использовать бинарную предикатную функцию по умолчанию, указанную в определении, то есть `std::ranges::less`, — элементы будут отсортированы в порядке возрастания. Функциональный объект `less` сравнивает два аргумента и возвращает `true`, если первый меньше второго. Если `less` возвращает `false`, то два значения зарплаты не расположены в порядке возрастания, поэтому алгоритм изменяет порядок соответствующих объектов `Employee`.
- Последний аргумент определяет проекцию. Это унарная функция, получающая элемент из диапазона и возвращающая часть полученного элемента. Здесь мы реализовали унарную функцию в виде лямбды, возвращающей значение зарплаты из указанного в аргументе объекта `Employee`. Проекция применяется до сравнения элементов, поэтому для определения порядка сортировки объектов `Employee` алгоритм `sort` сравнивает их не полностью, а только по значениям зарплаты.

```
43    // Сортируем объекты Employee по зарплате; обозначение {} сообщает
44    // алгоритму, что надо использовать функцию сравнения по умолчанию
45    std::ranges::sort(employees, {},
46       [](const auto& e) {return e.getSalary();});
47    std::cout << "\nEmployees sorted in ascending order by salary:\n";
48    std::ranges::copy(employees, output);
49
```

```
Employees sorted in ascending order by salary:
Indigo    Matthew   3587
Red       Jason     5000
Green     Ashley    7600
```

Сокращенное обозначение проекции

Мы можем заменить унарную функцию, реализованную в виде лямбды в строке 46, сокращенным обозначением:

```
&Employee::getSalary
```

Это создает указатель на функцию getSalary класса Employee, и мы можем записать строки 45–46 в сокращенном виде:

```
std::ranges::sort(employees, {}, &Employee::getSalary);
```

Чтобы функцию можно было использовать в качестве проекции, она должна быть открытой (public) и не должна иметь перегруженных версий. Кроме того, у нее не должно быть параметров, потому что алгоритмы std::ranges не могут получать дополнительные аргументы для их передачи в функцию класса, указанную в проекции[1].

Проекция может быть указателем на открытый элемент данных

Если класс содержит открытый элемент данных, вы можете передать указатель на него в качестве аргумента проекции. Например, если у класса Employee будет открытый элемент данных с именем salary, можно указать аргумент алгоритма sort следующим образом:

```
&Employee::salary
```

Проекция для сортировки array<Employee> в порядке убывания

В таких алгоритмах, как sort, принимающих функции в качестве аргументов, вы можете комбинировать пользовательские функции и проекции. Например, в строках 51–52 указаны и бинарный предикатный функциональный объект, и проекция для сортировки объектов Employee в порядке убывания зарплаты сотрудников. Мы снова передаем три аргумента:

- Первый аргумент (employees) — диапазон для сортировки.
- Второй аргумент создает функциональный объект std::ranges::greater, поэтому sort выполняет сортировку элементов в порядке убывания. Функциональный объект greater сравниваст два аргумента и возвращает true, если первый больше второго. Если greater возвращает false, два значения зарплаты не расположены в порядке убывания, поэтому алгоритм sort изменяет порядок соответствующих объектов Employee.
- Последний аргумент — проекция. В этом вызове функциональный объект std::ranges::greater сравниваст значения зарплаты для определения порядка сортировки.

```
50    // Сортируем объекты Employee по зарплате в обратном порядке
51    std::ranges::sort(employees, std::ranges::greater{},
52       &Employee::getSalary);
53    std::cout << "\nEmployees sorted in descending order by salary:\n";
54    std::ranges::copy(employees, output);
55 }
```

[1] «Под капотом» компилятора алгоритм std::ranges::sort использует функцию std::invoke (заголовок <functional>), чтобы вызывать предоставленный компаратор для каждого объекта Employee.

```
Employees sorted in descending order by salary:
Green    Ashley    7600
Red      Jason     5000
Indigo   Matthew   3587
```

14.7. Представления (C++20) и функциональное программирование

В разделе 6.14.3 вы познакомились с **представлениями (views)**, предназначенными для операций с диапазонами. Мы показали, что представления можно комбинировать, в том числе соединять их последовательно для обработки элементов диапазона в конвейере операций. Представление не имеет собственной копии элементов диапазона, оно просто перемещает элементы по конвейеру операций. Представления относятся к ключевым средствам функционального программирования C++20.

Алгоритмы, рассмотренные в этой главе, являются **жадными (greedy)** — когда вы их вызываете, они немедленно выполняют поставленную задачу. Представления, как было показано в разделе 6.14.3, являются **ленивыми (lazy)** — они не выдают результатов, пока вы не обработаете их в цикле или не передадите их алгоритму для обработки. Как мы рассказывали в разделе 6.14.3, ленивые вычисления выдают значения по требованию, снижая потребление памяти вашей программой и повышая быстродействие, когда не нужны все значения сразу.

14.7.1. Адаптеры диапазонов

В разделе 6.14.3 также было продемонстрировано функциональное программирование с использованием:

- `std::views::filter` — для сохранения только тех элементов представления, для которых предикатная функция возвращает `true`;
- `std::views::transform` — для преобразования каждого элемента представления в новое значение, которое может быть другого типа.

Это **адаптеры диапазонов (range adaptors)**. Представление (`view`) похоже на окно, которое позволяет вам «смотреть» в диапазон и наблюдать за его элементами. Представление не имеет собственной копии этих элементов. Каждый адаптер диапазонов принимает в качестве аргумента диапазон `std::ranges::viewable_range` и возвращает представление этого диапазона для использования в конвейере операций (мы рассказали о конвейерах в разделе 6.14). Концепт `viewable_range` обозначает такой диапазон, который можно безопасно преобразовать в представление[1]. Поскольку представления не владеют данными, которые они «видят», временный объект не может быть преобразован в `viewable_range`.

[1] «std::ranges::viewable_range». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/ranges/viewable_range.

В следующей таблице перечислены многие адаптеры диапазонов C++20 (заголовок `<ranges>`)[1]. Они позволяют применять методы функционального программирования, представленные в разделе 6.14. Адаптеры диапазонов определены в пространстве имен `std::ranges::views`, а также доступны через псевдоним `std::views`.

Адаптер диапазонов	Описание
`filter`	Создает представление, содержащее только те элементы диапазона, для которых предикат возвращает `true`
`transform`	Создает представление, преобразующее элементы в новые значения
`common`	Преобразует представление в `std::ranges::common_range`, что дает совместимость диапазона с алгоритмами, которым нужны пары итераторов `begin` — `end` одного и того же типа
`all`	Создает представление, содержащее все элементы диапазона
`counted`	Создает представление, содержащее указанное количество элементов с начала диапазона или с указанной позиции итератора
`reverse`	Создает представление для обработки двунаправленного (bidirectional) представления в обратном порядке
`drop`	Создает представление, игнорирующее указанное количество элементов от начала другого представления
`drop_while`	Создает представление, игнорирующее элементы другого представления от начала до тех пор, пока предикат возвращает `true`
`take`	Создает представление, содержащее указанное количество элементов от начала другого представления
`take_while`	Создает представление, содержащее элементы другого представления от начала до тех пор, пока предикат возвращает `true`
`join`	Создает представление, объединяющее элементы нескольких диапазонов
`split`	Разделяет представление на поддиапазоны, используя указанное значение в качестве разделителя
`keys`	Создает представление ключей из пар «ключ — значение», например из пар `map` (ключом является первый элемент в паре)
`values`	Создает представление значений из пар «ключ — значение», например из пар `map` (значением является второй элемент в паре)
`elements`	Для представления, содержащего кортежи (`tuple`), пары (`pair`) или массивы (`array`), создает представление, содержащее только элементы каждого объекта, имеющие указанный индекс

[1] «Ranges Library». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/ranges.

14.7.2. Работа с адаптерами диапазонов и представлениями

На илл. 14.18[1] демонстрируются несколько представлений `std::views` из предыдущей таблицы, а также версия представления `std::views::iota` для бесконечного диапазона. В строке 13 определяется лямбда, которая возвращает `true`, если ее аргумент является четным целым числом. Мы будем использовать эту лямбду в наших конвейерах.

```
 1 // fig14_18.cpp
 2 // Работа с std::views (C++20)
 3 #include <algorithm>
 4 #include <iostream>
 5 #include <iterator>
 6 #include <map>
 7 #include <ranges>
 8 #include <string>
 9 #include <vector>
10
11 int main() {
12    std::ostream_iterator<int> output{std::cout, " "};
13    auto isEven{[](int x) {return x % 2 == 0;}}; // true, если x - четное
14
```

Илл. 14.18 | Работа с std::views (C++20)

Создание бесконечного диапазона с помощью std::views::iota

На илл. 6.13 мы используем представление `std::views::iota`, известное как **фабрика диапазонов (range factory)**. Это представление, которое лениво создает последовательность возрастающих целых чисел по мере того, как вы выполняете итерации по этой последовательности. Показанная на илл. 6.13 в главе 6 версия `iota` требует два аргумента: первый — начальное целое число, второй — число, на `1` большее, чем конец последовательности, которую надо генерировать. В строке 16 илл. 14.18 используется версия `iota` для **бесконечного диапазона (infinite range)**, которая принимает только начальное целое число (`0`) в последовательности и увеличивает эту последовательность до тех пор, пока вы ее не остановите (вы увидите, что числа возрастают очень быстро). Конвейер в строке 16 создает представление, которое фильтрует целые числа, созданные `iota`, сохраняя только те из них, для которых лямбда-выражение `isEven` возвращает `true`. На данный момент никакая последовательность еще не создается. Напомним, что представления являются ленивыми: они не выполняются, пока вы не выполните итерации по ним с помощью цикла или алгоритма стандартной библиотеки. Представления — это объекты, которые вы можете хранить в переменных для повторного использования этапов обработки, чтобы добавлять дополнительные этапы обработки позже. Мы будем

[1] В январе 2022 г. эта программа не компилировалась в последней версии компилятора clang++.

хранить это представление в переменной evens и использовать ее для создания нескольких расширенных конвейеров.

```
15    // Создаем представление бесконечного диапазона четных чисел начиная с 0
16    auto evens{std::views::iota(0) | std::views::filter(isEven)};
17
```

Адаптер диапазонов take

Хотя бесконечный диапазон логически бесконечен[1], для его обработки в цикле или для его передачи в алгоритм стандартной библиотеки вы должны ограничить количество элементов, генерируемых конвейером; в противном случае в вашей программе появится бесконечный цикл. Один из способов — использовать адаптер диапазонов, ограничивающий количество обрабатываемых элементов. Такие адаптеры работают с бесконечными диапазонами и с диапазонами фиксированного размера. Например, в строке 19 используется адаптер диапазонов `take` для получения только первых пяти значений из конвейера `evens`, определенного в строке 16. Мы передаем полученное представление алгоритму `std::ranges::copy` (строка 19), выполняющему итерации в конвейере со следующими шагами:

- `iota` создает целое число;
- `filter` проверяет, является ли созданное число четным;
- если да, то `take` передаст его в алгоритм `copy`.

Если число, полученное от `iota`, нечетное, `filter` его отбрасывает, а `iota` выдает следующее число из последовательности. Этот процесс повторяется, пока адаптер `take` не передаст указанное количество элементов в алгоритм `copy`. Вы можете взять любое количество элементов из бесконечного диапазона или любое количество (в том числе максимальное) из диапазона фиксированного размера. В целях демонстрации мы обработаем только небольшое количество элементов в каждом из рассматриваемых конвейеров.

```
18    std::cout << "First five even ints: ";
19    std::ranges::copy(evens | std::views::take(5), output);
20
```

```
First five even ints: 0 2 4 6 8
```

Адаптер диапазонов take_while

В строках 22–23 создается расширенный конвейер, использующий адаптер диапазонов `take_while` для ограничения бесконечного диапазона. Этот адаптер возвращает представление, принимающее элементы из предыдущих этапов конвейера, пока унарный предикат адаптера `take_while` возвращает `true`. В нашем

[1] Джефф Гарланд (Jeff Garland), «Using C++20 Ranges Effectively», 18 июня 2019 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=VmWS-9idT3s.

примере мы берем четные целые числа, пока принимаемые значения меньше 12. Первое значение, большее или равное 12, завершает конвейер. Мы сохраняем представление, возвращаемое адаптером `take_while`, в переменной `lessThan12`, что позволяет использовать представление в последующих инструкциях. В строке 24 `lessThan12` передается алгоритму `std::ranges::copy`, который выполняет итерации по этому представлению, следуя шагам конвейера, и отображает результаты.

```
21    std::cout << "\nEven ints less than 12: ";
22    auto lessThan12{
23       evens | std::views::take_while([](int x) {return x < 12;})};
24    std::ranges::copy(lessThan12, output);
25
```

```
Even ints less than 12: 0 2 4 6 8 10
```

Адаптер диапазонов reverse

В строке 27 адаптер диапазонов `reverse` расставляет в обратном порядке целые числа представления `lessThan12` из строк 22–23. Представление, возвращенное адаптером `reverse`, мы передаем алгоритму `std::ranges::copy`, который выполняет итерации по этому представлению, следуя шагам конвейера, и отображает результаты.

```
26    std::cout << "\nEven ints less than 12 reversed: ";
27    std::ranges::copy(lessThan12 | std::views::reverse, output);
28
```

```
Even ints less than 12 reversed: 10 8 6 4 2 0
```

Адаптер диапазонов transform

Вы познакомились с адаптером диапазонов `transform` на илл. 6.13. Конвейер в строках 31–33 создает представление, получающее целые числа, созданные `lessThan12`, расставляет их в обратном порядке и использует адаптер `transform` для возведения каждого значения в квадрат. Мы передаем итоговое представление алгоритму `std::ranges::copy`, который выполняет итерации по этому представлению, следуя шагам конвейера, и отображает результаты.

```
29    std::cout << "\nSquares of even ints less than 12 reversed: ";
30    std::ranges::copy(
31       lessThan12
32          | std::views::reverse
33          | std::views::transform([](int x) {return x * x;}),
34       output);
35
```

```
Squares of even ints less than 12 reversed: 100 64 36 16 4 0
```

Адаптер диапазонов drop

Конвейер в строке 38 начинает с бесконечной последовательности четных целых чисел, созданных переменной `evens`, использует адаптер диапазонов `drop` для пропуска первых `1000` четных целых чисел, а затем адаптер `take` для получения следующих пяти четных целых чисел из последовательности. Мы передаем итоговое представление алгоритму `std::ranges::copy`, который выполняет итерации по этому представлению, следуя шагам конвейера, и отображает результаты.

```
36    std::cout << "\nSkip 1000 even ints, then take five: ";
37    std::ranges::copy(
38       evens | std::views::drop(1000) | std::views::take(5),
39       output);
40
```

```
Skip 1000 even ints, then take five: 2000 2002 2004 2006 2008
```

Адаптер диапазонов drop_while

Вы также можете пропускать элементы, пока унарный предикат возвращает `true`. Конвейер в строках 43–45 начинается с бесконечной последовательности четных целых чисел, находящихся в `evens`. Сначала адаптер диапазонов `drop_while` используется для пропуска четных целых чисел из начала бесконечной последовательности, которые меньше или равны `1000`. Затем адаптер `take` используется для получения следующих пяти четных целых чисел. Мы передаем итоговое представление алгоритму `std::ranges::copy`, который выполняет итерации по этому представлению, следуя шагам конвейера, и отображает результаты.

```
41    std::cout << "\nFirst five even ints greater than 1000: ";
42    std::ranges::copy(
43       evens
44          | std::views::drop_while([](int x) {return x <= 1000;})
45          | std::views::take(5),
46       output);
47
```

```
First five even ints greater than 1000: 1002 1004 1006 1008 1010
```

Создание и отображение контейнера map, содержащего римские цифры и десятичные эквиваленты

Для удобства обсуждения мы фокусировались на обработке диапазонов, состоящих из целых чисел, но вы можете обрабатывать диапазоны, содержащие более сложные типы данных, и обрабатывать различные контейнеры. Далее мы обработаем пары «ключ — значение» в контейнере `map`, содержащем объекты `string` в качестве ключей и объекты `int` в качестве значений. Ключи — это римские цифры, а значения — соответствующие им десятичные значения. Объявление

`using` в строке 49 позволяет нам использовать строковые объектные литералы при построении каждой пары «ключ — значение» в строках 51–52. Например, в определении `"I"s` буква `s`, следующая за строковым литералом, обозначает, что литерал является объектом `string`. В строках 53–54 определено лямбда-выражение, которое мы используем в строке 56 в алгоритме `std::ranges::for_each` для отображения каждой пары «ключ — значение» из контейнера `map`.

```
48    // Используем объектные литералы std::string
49    using namespace std::string_literals;
50
51    std::map<std::string, int> romanNumerals{
52       {"I"s, 1}, {"II"s, 2}, {"III"s, 3}, {"IV"s, 4}, {"V"s, 5}};
53    auto displayPair{[](const auto& p) {
54       std::cout << p.first << " = " << p.second << "\n";}};
55    std::cout << "\n\nromanNumerals:\n";
56    std::ranges::for_each(romanNumerals, displayPair);
57
```

```
romanNumerals:
I = 1
II = 2
III = 3
IV = 4
V = 5
```

Адаптеры диапазонов keys и values

При работе с контейнером `map` в конвейерах каждая пара «ключ — значение» обрабатывается как объект `pair`, содержащий ключ и значение. Вы можете использовать адаптеры диапазонов `keys` (строка 60) и `values` (строка 63), чтобы получить представления ключей или значений соответственно:

- адаптер `keys` создает представление, содержащее только первые элементы объектов `pair`;
- адаптер `values` создает представление, содержащее только вторые элементы объектов `pair`.

Мы передаем каждый конвейер алгоритму `std::ranges::copy`, который выполняет итерации по конвейеру и отображает результаты.

```
58    std::ostream_iterator<std::string> stringOutput{std::cout, " "};
59    std::cout << "\nKeys in romanNumerals: ";
60    std::ranges::copy(romanNumerals | std::views::keys, stringOutput);
61
62    std::cout << "\nValues in romanNumerals: ";
63    std::ranges::copy(romanNumerals | std::views::values, output);
64
```

```
Keys in romanNumerals: I II III IV V
Values in romanNumerals: 1 2 3 4 5
```

Адаптер диапазонов elements

Интересно, что адаптеры диапазонов `keys` и `values` также работают с диапазонами, в которых каждый элемент — это объект `tuple` или `array`. Даже если объекты содержат более двух элементов, адаптер `keys` всегда выбирает первый элемент, а `values` всегда выбирает второй элемент. Если объекты `tuple` или `array` в диапазоне содержат более двух элементов, адаптер диапазонов `elements` может выбирать элементы по индексу, как в нашем примере с парами «ключ — значение» `romanNumerals`. В строке 67 выбирается только элемент с индексом `0` из каждого объекта `pair` в диапазоне, а в строке 70 выбирается только элемент с индексом `1`. В обоих случаях мы передаем конвейер алгоритму `std::ranges::copy`, который выполняет итерации по конвейерам и отображает результаты.

```
65    std::cout << "\nKeys in romanNumerals via std::views::elements: ";
66    std::ranges::copy(
67       romanNumerals | std::views::elements<0>, stringOutput);
68
69    std::cout << "\nvalues in romanNumerals via std::views::elements: ";
70    std::ranges::copy(romanNumerals | std::views::elements<1>, output);
71    std::cout << "\n";
72 }
```

```
Keys in romanNumerals via std::views::elements: I II III IV V
values in romanNumerals via std::views::elements: 1 2 3 4 5
```

14.8. Основы параллельных алгоритмов

На протяжении десятилетий за каждые два года лучшие промышленно выпускаемые процессоры становились примерно вдвое мощнее. Эту тенденцию назвали законом Мура — в честь Гордона Мура, соучредителя Intel, который заметил ее еще в 1960-х. Топ-менеджеры компаний NVIDIA и Arm, выпускающих процессоры, заявляют, что закон Мура больше не действует[1,2]. Вычислительная мощность компьютеров продолжает расти, но производители оборудования делают упор на такие новые технологии, как многоядерные процессоры, выполняющие параллельные операции для повышения быстродействия. C++ всегда был ориентирован на максимальное быстродействие, но поддержка параллелизма впервые была реализована только в C++11, через 32 года после создания языка.

Распараллелить алгоритм не так просто, как «щелкнуть пальцами» и сказать: «Я хочу запустить этот алгоритм параллельно». Распараллеливание зависит от внутренней реализации алгоритма и от наличия процессов, которые действительно

[1] Эстер Шейн (Esther Shein), «Moore's Law Turns 55: Is It Still Relevant?», 17 апреля 2020 г. Был доступен: 15 сентября 2023 г. https://www.techrepublic.com/article/moores-law-turns-55-is-it-still-relevant.

[2] Ник Хит (Nick Heath), «Moore's Law Is Dead: Three Predictions About the Computers of Tomorrow», 19 сентября 2018 г. Был доступен: 15 сентября 2023 г. https://www.techrepublic.com/article/moores-law-is-dead-three-predictions-about-the-computers-of-tomorrow/.

могут выполняться параллельно на многоядерном оборудовании. Программисты должны тщательно определять задачи, которые могут выполняться параллельно. Такие алгоритмы должны уметь масштабироваться на любое количество ядер: в любой момент времени может быть доступно больше или меньше ядер, поскольку компьютер распределяет ядра между всеми своими задачами. Кроме того, количество ядер в выпускаемых процессорах увеличивается по мере развития компьютерной архитектуры, и алгоритмы должны быть достаточно гибкими, чтобы использовать преимущества этих дополнительных ядер. Несмотря на всю сложность создания подобных алгоритмов, мы должны стараться увеличивать быстродействие приложений.

С годами стало понятно, что алгоритмы, способные выполнять задачи на нескольких ядрах, сложны в разработке и подвержены ошибкам. В настоящее время во многих языках программирования есть встроенные библиотеки, предоставляющие программисту удобные функции для выполнения параллельных операций. В C++ уже есть набор мощных алгоритмов для этой цели. Чтобы не приходилось «изобретать колесо», в C++17 были добавлены перегруженные параллельные версии 69 алгоритмов для работы с обычными диапазонами, что позволяет пользоваться преимуществами многоядерных архитектур и быстродействующих операций «векторной математики», доступных на современных центральных и графических процессорах. Векторные операции могут выполнять одну и ту же операцию со многими элементами данных одновременно[1].

Кроме того, в C++17 было добавлено семь новых параллельных алгоритмов, каждый из которых также имеет и обычную последовательную версию:

- `for_each_n`
- `exclusive_scan`
- `inclusive_scan`
- `transform_exclusive_scan`
- `transform_inclusive_scan`
- `reduce`
- `transform_reduce`

Перегрузки алгоритмов в значительной степени снимают бремя распараллеливания задач с ваших плеч и возлагают его на стандартную библиотеку C++, повышая эффективность программирования. К сожалению, алгоритмы из пространства имен `std::ranges` (C++20) еще не имеют перегруженных параллельных версий, но они могут появиться в C++23[2].

---

[1] «General Utilities Library — Execution Policies — Unsequenced Execution Policy». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/execpol.unseq.

[2] Барри Ревзин (Barry Revzin), Конор Хукстра (Conor Hoekstra) и Тим Сонг (Tim Song), «A Plan for C++23 Ranges», 14 октября 2020 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p2214r0.

В главе 17 мы:

- рассмотрим параллельные алгоритмы;
- обсудим четыре политики выполнения (три из C++17 и одну из C++20), которые определяют, каким образом параллельный алгоритм использует для выполнения задачи средства параллельной обработки, имеющиеся в операционной системе;
- объясним, как вызывать параллельные алгоритмы;
- продемонстрируем функции из заголовка `<chrono>`, сравнивая время выполнения алгоритмов стандартной библиотеки, работающих последовательно и параллельно, чтобы вы увидели разницу в быстродействии.

Вы увидите, что параллельные версии алгоритмов не всегда выполняются быстрее, чем последовательные. Мы объясним причины этого. Мы также представим несколько заголовков стандартной библиотеки C++, содержащих средства параллельного программирования и параллелизма. Затем, в главе 18, мы представим введенные в C++20 корутины, также называемые сопроцедурами, или сопрограммами.

14.9. Краткое описание алгоритмов стандартной библиотеки

В стандарте C++ определено 117 алгоритмов, многие из которых имеют две и более перегруженные версии. Стандарт разделяет алгоритмы на несколько категорий:

- алгоритмы, модифицирующие последовательности (`<algorithm>`);
- алгоритмы, не модифицирующие последовательности (`<algorithm>`);
- алгоритмы сортировки и сравнения (`<algorithm>`);
- алгоритмы для числовых операций (`<numeric>`);
- специализированные алгоритмы для работы с памятью (`<memory>`).

Информацию об алгоритмах, не представленных в этой главе, вы найдете здесь:

https://en.cppreference.com/w/cpp/algorithm

https://docs.microsoft.com/ru-ru/cpp/standard-library/algorithm

Во всех таблицах этого раздела:

- Алгоритмы, представленные в этой главе, сгруппированы в верхней части каждой таблицы.
- Алгоритмы, имеющие перегруженную версию в пространстве имен `std::ranges` (C++20), отмечены символом «R»[1].

[1] «Constrained Algorithms». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/algorithm/ranges.

- Алгоритмы, имеющие параллельную версию, отмечены символом «P»[1].
- Алгоритмы, введенные в C++11, C++17 и C++20, отмечены номером версии.

Например, алгоритм `copy` отмечен символами «PR», то есть он имеет параллельную версию и перегруженную версию в пространстве имен `std::ranges`. Алгоритм `is_sorted_until` отмечен знаком «PR11», то есть он имеет параллельную версию и перегруженную версию в пространстве имен `std::ranges`; он был введен в стандартную библиотеку в C++11.

Алгоритмы, модифицирующие последовательности

В следующей таблице перечислены многие алгоритмы, модифицирующие последовательности, то есть алгоритмы, изменяющие контейнеры, с которыми они работают. Алгоритм `shuffle` был добавлен вместо менее безопасного алгоритма `random_shuffle`. Алгоритм `random_shuffle` использовал «под капотом» функцию `rand`, унаследованную от стандартной библиотеки языка C. Эта функция не обладает «хорошими статистическими свойствами» и может быть предсказуемой[2], что плохо влияет на безопасность программ. Новый алгоритм `shuffle` основан на более надежных средствах недетерминированной генерации случайных чисел, введенных в C++11.

Алгоритмы, модифицирующие последовательности (заголовок <algorithm>)			
`copy` (PR)	`copy_backward` (R)	`copy_if` (PR11)	`copy_n` (PR11)
`fill` (PR)	`fill_n` (PR)	`generate` (PR)	`generate_n` (PR)
`iter_swap`	`remove` (PR)	`remove_copy` (PR)	`remove_copy_if` (PR)
`remove_if` (PR)	`replace` (PR)	`replace_copy` (PR)	`replace_copy_if` (PR)
`replace_if` (PR)	`reverse` (PR)	`reverse_copy` (PR)	`shuffle` (R11)
`swap_ranges` (PR)	`transform` (PR)	`unique` (PR)	`unique_copy` (PR)
`move` (PR11)	`move_backward` (R11)	`rotate` (PR)	`rotate_copy` (PR)
`sample` (R17)	`shift_left` (20)	`shift_right` (20)	

[1] «Extensions for Parallelism». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/experimental/parallelism/. (Хотя в заголовках на этой странице есть слово «experimental», эти параллельные версии официально являются частью стандарта C++.)

[2] «Do Not Use the rand() Function for Generating Pseudorandom Numbers». Последнее изменение: 23 апреля 2021 г. Был доступен: 15 сентября 2023 г. https://wiki.sei.cmu.edu/confluence/display/c/MSC30-C.+Do+not+use+the+rand()+function+for+generating+pseudorandom+numbers/.

Алгоритмы, не модифицирующие последовательности

В следующей таблице перечислены алгоритмы, не модифицирующие последовательности, то есть алгоритмы, не изменяющие контейнеры, с которыми они работают.

Алгоритмы, не модифицирующие последовательности (заголовок <algorithm>)			
all_of (PR11)	**any_of** (PR11)	**count** (PR)	**count_if** (PR)
equal (PR)	**find** (PR)	**find_if** (PR)	**find_if_not** (PR11)
for_each (PR)	**mismatch** (PR)	**none_of** (PR11)	
adjacent_find (PR)	**find_end** (PR)	**find_first_of** (PR)	**for_each_n** (PR17)
is_permutation (R11)	**search** (PR)	**search_n** (PR)	

Алгоритмы сортировки и сравнения

В следующей таблице перечислены алгоритмы сортировки и сравнения.

Алгоритмы сортировки и сравнения (заголовок <algorithm>)		
binary_search (R)	**equal_range** (R)	**includes** (PR)
inplace_merge (PR)	**lexicographical_compare** (PR)	**lower_bound** (R)
make_heap (R)	**max** (R)	**max_element** (PR)
merge (PR)	**min** (R)	**min_element** (PR)
minmax (R11)	**minmax_element** (PR11)	**pop_heap** (R)
push_heap (R)	**set_difference** (PR)	**set_intersection** (PR)
set_symmetric_difference (PR)	**set_union** (PR)	**sort** (PR)
sort_heap (R)	**upper_bound** (R)	
clamp (R17)	**is_heap** (PR11)	**is_heap_until** (PR11)
is_partitioned (PR11)	**is_sorted** (PR11)	**is_sorted_until** (PR11)
lexicographical_compare_three_way (20)		**next_permutation** (R)
nth_element (PR)	**partial_sort** (PR)	**partial_sort_copy** (PR)
partition (PR)	**partition_copy** (PR11)	**partition_point** (R11)
prev_permutation (R)	**stable_partition** (PR)	**stable_sort** (PR)

Алгоритмы для операций с числами

В следующей таблице перечислены алгоритмы из заголовка `<numeric>` для числовых операций. Алгоритмы из этого заголовка еще не имеют перегруженных версий для работы с `std::ranges` (C++20), но эти версии могут появиться в C++23[1,2].

Алгоритмы для операций с числами (заголовок <numeric>)		
accumulate	**gcd** (17)	**iota** (11)
`lcm` (17)	**partial_sum**	**reduce** (P17)
`adjacent_difference` (P)	`exclusive_scan` (P17)	`inclusive_scan` (P17)
`inner_product` (P)	`midpoint` (20)	`transform_reduce` (P17)
`transform_exclusive_scan` (P17)	`transform_inclusive_scan` (P17)	

Специализированные операции для работы с памятью

В следующей таблице показаны специализированные алгоритмы из заголовка `<memory>` для управления динамической памятью. Их рассмотрение выходит за рамки этой книги. Здесь вы найдете базовую информацию об этом заголовке и его алгоритмах:

https://en.cppreference.com/w/cpp/header/memory

Специализированные операции для работы с памятью (заголовок <memory>)	
`construct_at` (R20)	`destroy` (R17)
`destroy_at` (R17)	`destroy_n` (R17)
`uninitialized_copy` (PR)	`uninitialized_copy_n` (PR11)
`uninitialized_default_construct` (PR17)	`uninitialized_default_construct_n` (PR17)
`uninitialized_fill` (PR)	`uninitialized_fill_n` (PR)
`uninitialized_move` (PR17)	`uninitialized_move_n` (PR17)
`uninitialized_value_construct` (PR17)	`uninitialized_value_construct_n` (PR17)

[1] Кристофер Ди Белла (Christopher Di Bella), «A Concept Design for the Numeric», 2 августа 2019 г. Был доступен: 15 сентября 2023 г. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1813r0.pdf.

[2] Тристан Бриндл (Tristan Brindle), «Numeric Range Algorithms for C++20», 19 мая 2020 г. Был доступен: 15 сентября 2023 г. https://tristanbrindle.com/posts/numeric-ranges-for-cpp20.

14.10. Диапазоны в C++23

Хотя многие алгоритмы имеют перегруженные версии в пространстве имен `std::ranges` (C++20), многие введенные в C++20 алгоритмы (в том числе алгоритмы из заголовка `<numeric>` и параллельные алгоритмы из заголовка `<algorithm>`) еще не перегружены в пространстве имен `std::ranges`. Есть предложения Комитета по стандартам, относящиеся к различным дополнительным функциям библиотеки `ranges`, которые могут стать частью C++23. Некоторые из этих возможностей пока находятся в стадии разработки, но их можно опробовать с помощью проекта с открытым исходным кодом `rangesnext` («Ranges for C++23»)[1]. Функциональность диапазонов C++20 и многие новые средства, предлагаемые для C++23, основаны на проекте с открытым исходным кодом `range-v3`[2].

«План для диапазонов C++23»[3]

В этом документе вы найдете обзор основного плана для диапазонов C++23, а также подробные сведения о предлагаемых новых средствах диапазонов. Предлагаемые средства разделены по степени важности на три уровня, причем первый уровень содержит наиболее важные из них. После краткого введения в этом документе обсуждаются несколько категорий возможных дополнений:

- «View adjuncts». В этом разделе кратко рассматриваются два ключевых средства: перегруженная функция `ranges::to` для преобразования представлений в различные типы контейнеров и средство форматирования представлений и диапазонов для удобного вывода. Они подробно обсуждаются в публикациях «ranges::to: A Function to Convert Any Range to a Container»[4] и «Formatting Ranges»[5] соответственно.
- «Algorithms». В этом разделе рассматриваются предлагаемые новые версии алгоритмов из заголовка `<numeric>`, перегруженные в пространстве имен `std::ranges`, которые более подробно рассматриваются в документе «A Concept Design for the Numeric Algorithms»[6]. В этом разделе также кратко рассматриваются потенциальные проблемы с перегруженными в `std::ranges`

---

[1] Корентен Жабо (Corentin Jabot), «Ranges for C++23». Был доступен: 15 сентября 2023 г. https://github.com/cor3ntin/rangesnext.

[2] Эрик Ниблер (Eric Niebler), «range-v3». Был доступен: 15 сентября 2023 г. https://github.com/ericniebler/range-v3.

[3] Барри Ревзин (Barry Revzin), Конор Хукстра (Conor Hoekstra) и Тим Сонг (Tim Song), «A Plan for C++23 Ranges», 14 октября 2020 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p2214r0.

[4] Корентен Жабо (Corentin Jabot), Эрик Ниблер (Eric Niebler) и Кейси Картер (Casey Carter), «ranges::to: A Function to Convert Any Range to a Container», 22 ноября 2021 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p1206r3.

[5] Барри Ревзин (Barry Revzin), «Formatting Ranges», 19 февраля 2021 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p2286.

[6] Кристофер Ди Белла (Christopher Di Bella), «A Concept Design for the Numeric Algorithms», 2 августа 2019 г. Был доступен: 15 сентября 2023 г. http://wg21.link/p1813r0.

версиями параллельных алгоритмов C++17. В документе «Introduce Parallelism to the Ranges TS»[1] содержится более подробное обсуждение этих вопросов.

- «Actions». Это третья категория предлагаемых новинок языка C++, независимая от диапазонов и представлений в проекте range-v3. Действия (actions), как и представления, можно комбинировать с помощью оператора |, но, подобно алгоритмам `std::ranges`, действия являются жадными, то есть выдают результаты немедленно. Согласно этому разделу, действия могут сделать программирование более удобным, но их добавление имеет низкий приоритет, поскольку те же задачи можно выполнять, комбинируя существующие алгоритмы `std::ranges` в нескольких инструкциях.

14.11. Итоги

В этой главе мы рассмотрели многие алгоритмы стандартной библиотеки, позволяющие заполнять контейнеры значениями, генерировать значения, сравнивать отдельные элементы или целые контейнеры, удалять или заменять элементы, выполнять арифметические вычисления, операции поиска, сортировки, копирования, объединения и сравнения множеств; определять границы и находить минимальные и максимальные элементы. Мы сосредоточились в первую очередь на версиях этих алгоритмов из пространства имен `std::ranges`.

Вы узнали, что алгоритмы стандартной библиотеки определяют различные минимальные требования, которые помогают понять, какие контейнеры, итераторы и функции можно передавать разным алгоритмам. Мы дали общее представление о некоторых требованиях алгоритмов обычных диапазонов, затем показали, что алгоритмы диапазонов C++20 используют концепты C++20 для определения своих требований, проверяемых во время компиляции. Мы кратко рассмотрели концепты, указанные для каждого представленного нами алгоритма диапазонов C++20. Не каждый алгоритм имеет перегруженную версию для диапазонов C++20.

Мы углубились в лямбда-выражения и познакомили вас с их дополнительными возможностями для захвата переменных из внешней области видимости. Вы узнали, что многие алгоритмы могут получать в качестве аргумента лямбду, указатель на функцию или функциональный объект и вызывать их для настройки поведения алгоритмов.

Мы продолжили обсуждение функционального программирования на C++. Мы показали, как создать логически бесконечную последовательность значений и как использовать адаптеры диапазонов для ограничения количества элементов, обрабатываемых в конвейере. Вы научились сохранять представления в переменных для последующего использования и добавлять следующие шаги к ранее

---

[1] Гордон Браун (Gordon Brown), Кристофер Ди Белла, Майкл Хайдл (Michael Haidl), Тоомас Реммельг (Toomas Remmelg), Руйман Рейес (Ruyman Reyes), Мишель Штювер (Michel Steuwer) и Майкл Вонг (Michael Wong), «P0836R1 Introduce Parallelism to the Ranges TS», 7 мая 2018 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p0836r1.

сохраненному конвейеру. Мы представили адаптеры диапазонов для обработки ключей и значений в парах «ключ — значение» и показали адаптер диапазонов для выбора любого индексированного элемента из объектов фиксированного размера, таких как `pair`, `tuple` и `array`.

Вы узнали, что в C++17 были добавлены новые параллельные перегруженные версии 69 алгоритмов стандартной библиотеки из заголовка `<algorithm>`. Как вы увидите в главе 17, они позволят вам использовать сразу несколько ядер многоядерного процессора вашего компьютера для повышения быстродействия программ. В главе 17 мы покажем несколько параллельных алгоритмов и используем возможности заголовка `<chrono>`, чтобы сравнить время выполнения последовательных и параллельных алгоритмов, показывая различия в быстродействии. Мы объясним, почему параллельные алгоритмы не всегда выполняются быстрее, чем их последовательные аналоги.

Вы узнали, что некоторые алгоритмы C++20, включая алгоритмы из заголовка `<numeric>` и параллельные алгоритмы из заголовка `<algorithm>`, еще не имеют перегруженных версий в пространстве имен `std::ranges`. Мы кратко рассмотрели обновления, ожидаемые в C++23, и упомянули опубликованный на GitHub проект `rangesnext`, содержащий реализации многих предлагаемых новинок.

В следующей главе мы создадим простой контейнер, итераторы и простой алгоритм с помощью пользовательских шаблонов, в том числе шаблонов с концептами C++20.

15

Шаблоны, концепты (C++20) и метапрограммирование

В этой главе вы:

- оцените преимущества обобщенного программирования;
- научитесь использовать шаблоны классов для создания связанных пользовательских классов;
- больше узнаете о динамическом и статическом полиморфизме;
- научитесь различать шаблоны и специализации шаблонов;
- научитесь использовать введенные в C++20 сокращенные шаблоны функций и шаблоны лямбда-выражений;;
- научитесь использовать концепты (C++20) для ограничения параметров шаблонов и для перегрузки шаблонов функций на основе их требований к типам;
- научитесь использовать признаки типов и узнаете, как они соотносятся с концептами;
- протестируете несколько концептов во время компиляции с помощью объявления `static_assert`;
- создадите пользовательский алгоритм, ограниченный концептами;
- переопределите класс `myArray` в виде пользовательского шаблона контейнера с пользовательскими итераторами;
- научитесь использовать нетипизированные параметры шаблона для передачи шаблонам констант, созданных на этапе компиляции, и использовать аргументы шаблона по умолчанию;
- научитесь использовать вариативные шаблоны, получающие произвольное количество параметров, и применять к ним бинарные операторы в выражениях свертки;
- попробуете свои силы в метапрограммировании, вычисляя значения, оперируя типами и генерируя код на этапе компиляции, чтобы повысить быстродействие программ.

15.1. Введение

Обобщенное программирование с применением шаблонов существует в C++ с 1998 года, когда был выпущен стандарт C++[1]. В каждой очередной версии стандарта эта технология становилась все более важной и совершенной. Современная тенденция C++ заключается в том, чтобы выполнять как можно больше вычислений на этапе компиляции с целью более надежной проверки типов и наибольшего быстродействия во время выполнения программы[2,3,4]. Вы уже много раз использовали шаблоны. Как вы скоро узнаете, шаблоны и метапрограммирование на основе шаблонов — это ключевые средства для выполнения многих сложных операций на этапе компиляции. В этой главе мы подробнее рассмотрим шаблоны, расскажем о разработке пользовательских шаблонов классов, изучим концепты (наиболее значимую новинку C++20) и представим технологию основанного на шаблонах метапрограммирования. В следующей таблице перечислены используемые нами шаблоны и связанные с ними темы в соответствующих главах книги.

Глава	Шаблоны и связанные с ними темы
Глава 1	Основы объектно-ориентированного программирования
Главы 2–4	Строки `string` (шаблон класса; `string` — это псевдоним `basic_string<char>`)
Глава 5	Раздел 5.8: шаблон класса `uniform_int_distribution` для генерации случайных чисел. Раздел 5.16: определение шаблона функции
Глава 6	Шаблоны классов контейнеров `array` и `vector` стандартной библиотеки
Глава 7	Раздел 7.10: шаблон класса `span` для создания представления контейнера смежных элементов, такого как `array` или `vector`
Глава 8	Разделы 8.2–8.9: углубленное изучение строк `string` (шаблон класса). Разделы 8.12–8.16: классы для обработки файловых потоков (шаблоны классов). Раздел 8.17: шаблоны классов для работы со строковыми потоками. Раздел 8.19: шаблоны функций библиотеки `rapidcsv` для обработки данных в формате CSV
Глава 9	Раздел 9.22: шаблоны функций библиотеки `cereal` для сериализации и десериализации данных в формате JSON

[1] «History of C++». Был доступен: 15 сентября 2023 г. https://www.cplusplus.com/info/history/.

[2] C++ Core Guidelines, «Per: Performance». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-performance.

[3] «Big Picture Issues – What's the Big Deal with Generic Programming?». Был доступен: 15 сентября 2023 г. https://isocpp.org/wiki/faq/big-picture#generic-paradigm.

[4] «Compile Time vs. Run Time Polymorphism in C++: Advantages/Disadvantages». Был доступен: 15 сентября 2023 г. https://stackoverflow.com/questions/16875989/compile-time-vs-run-time-polymorphism-in-c-advantages-disadvantages.

Глава	Шаблоны и связанные с ними темы
Глава 10	Раздел 10.13: полиморфизм на основе шаблона класса `std::variant` и шаблона функции `std::visit`
Глава 11	Умные указатели, управляющие динамически выделяемой памятью с помощью шаблона класса `unique_ptr`; шаблон функции `make_unique` и шаблон класса `initializer_list` для передачи списков инициализаторов в функции
Глава 12	Раздел 12.8: шаблон класса `unique_ptr`
Глава 13	Шаблоны классов контейнеров стандартной библиотеки и итераторы (реализованные как шаблоны классов)
Глава 14	Шаблоны функций-алгоритмов стандартной библиотеки, работающие с шаблонами классов контейнеров стандартной библиотеки с помощью итераторов
Глава 15	Пользовательские шаблоны классов, шаблоны итераторов, шаблоны функций, сокращенные шаблоны функций, шаблоны лямбда-выражений, признаки типов, концепты (C++20), перегрузка на основе концептов, шаблоны переменных, шаблоны-псевдонимы, вариативные шаблоны, выражения свертки и метапрограммирование на основе шаблонов
Глава 16	Параллельные алгоритмы стандартной библиотеки и другие шаблоны функций и классов, связанные с разработкой многопоточных приложений

Пользовательские шаблоны

В разделе 5.16 мы объяснили, как компилятор генерирует по шаблону функции множество перегруженных функций: это называется **инстанцированием (instantiating)**, или **созданием специализаций шаблона функции (instantiating the function template)**. Аналогично компилятор генерирует по шаблону класса множество похожих классов: это называется **созданием специализаций шаблона класса (instantiating the class template)**. На этом основаны методы статического полиморфизма, то есть полиморфизма на этапе компиляции. В этой главе мы создадим пользовательские шаблоны классов и рассмотрим ключевые технологии, связанные с ними.

Средства для работы с шаблонами, введенные в C++20

Мы рассмотрим новые средства работы с шаблонами, включая сокращенные шаблоны функций и шаблоны лямбда-выражений, а также концепты, входящие в «большую четверку» новинок C++20. Концепты упрощают и ускоряют обобщенное программирование на основе шаблонов. Когда вызывается любой из имеющихся в C++20 алгоритмов обработки диапазонов, все контейнеры и итераторы, передаваемые этому алгоритму в аргументах, должны соответствовать его требованиям. В этой главе мы рассмотрим шаблоны с точки зрения их разработчика и напишем пользовательские шаблоны, которые устанавливают собственные требования,

используя предопределенные и пользовательские концепты[1, 2, 3, 4]. Компилятор проверяет концепты перед созданием каждой специализации шаблона, поэтому когда программист ошибается, сообщения компилятора об этих ошибках становятся значительно конкретнее и понятнее, чем было прежде.

Признаки типов

Мы рассмотрим признаки типов, предназначенные для тестирования атрибутов встроенных и пользовательских типов классов во время компиляции. Вы увидите, что многие концепты реализованы в терминах признаков типов. Концепты упрощают использование признаков типов для ограничения параметров шаблонов.

Создание пользовательских контейнеров, итераторов и алгоритмов

Класс `myArray` из главы 11 был рассчитан только на хранение целых чисел. Теперь мы создадим более совершенный шаблон класса `myArray`, рассчитанный на хранение элементов различных типов (например, чисел с плавающей точкой или объектов класса `Employee`). Мы сделаем объекты класса `myArray` совместимыми со многими алгоритмами стандартной библиотеки, определив пользовательские итераторы. Мы также определим пользовательский алгоритм, обрабатывающий элементы контейнеров `myArray` и контейнеров стандартной библиотеки.

Вариативные шаблоны и выражения свертки

Мы создадим вариативный шаблон функции, получающий произвольное количество параметров. Мы также представим вам выражения свертки (C++17), позволяющие удобно применять операцию ко всем элементам, переданным вариативному шаблону.

Метапрограммирование на основе шаблонов

Согласно C++ Core Guidelines, метапрограммирование на основе шаблонов — это «создание программ, составляющих код во время компиляции»[5]. Компилятор также может использовать шаблоны для того, чтобы уже на этапе компиляции выполнять вычисления, а также операции с типами данных. Вычисления во время компиляции могут повысить быстродействие программы во время выполнения, например, путем уменьшения нагрузки на процессор или потребления памяти. Вы увидите, что в метапрограммировании на основе шаблонов широко

---

[1] Мариус Банчила (Marius Bancila), «Concepts Versus SFINAE-Based Constraints». Был доступен: 15 сентября 2023 г. https://mariusbancila.ro/blog/2019/10/04/concepts-versus-sfinae-based-constraints/.

[2] Саар Раз (Saar Raz), «C++20 Concepts: A Day in the Life», 17 октября 2019 г. https://www.youtube.com/watch?v=qawSiMIXtE4.

[3] «Constraints and Concepts». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/language/constraints.

[4] «Concepts Library». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/concepts.

[5] C++ Core Guidelines, «T: Templates and Generic Programming». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-templates.

используются **признаки типов (type traits)** для генерации кода в зависимости от атрибутов шаблона-аргумента. Мы напишем шаблон функции, генерирующий различные варианты кода во время компиляции в зависимости от того, поддерживает ли его аргумент итераторы произвольного доступа или только итераторы более низкого уровня. Вы увидите, как это позволяет ускорить работу программы во время выполнения.

До C++20 многие программисты C++ считали метапрограммирование на основе шаблонов слишком сложным для практического применения. Введенные в C++20 концепты облегчают эту работу. Тем не менее в Google C++ Style Guide рекомендуется «избегать сложного программирования на основе шаблонов»[1]. В C++ Core Guidelines указано, что вы должны «использовать метапрограммирование на основе шаблонов, только если это вам действительно необходимо» и что такой код «трудно сделать правильным... и чаще всего очень сложно поддерживать»[2]. Эти рекомендации могут измениться в будущем, когда программисты приобретут больше опыта работы с концептами и с дальнейшими усовершенствованиями, которые, возможно, появятся в C++23.

15.2. Пользовательские шаблоны классов и статический полиморфизм

Шаблон обеспечивает статический полиморфизм (полиморфизм во время компиляции)[3, 4], описывая требуемую функциональность класса в обобщенном виде и позволяя компилятору создавать специализации шаблона, то есть генерировать специализированные версии этого класса для обработки конкретных типов данных. Шаблоны классов называют параметризованными типами, поскольку им надо передавать один или несколько параметров, указывающих компилятору, как надо специализировать шаблон, чтобы получить из него класс, пригодный для создания объектов. Вы создаете единственное определение шаблона класса. Когда вам нужны его конкретные специализации, вы используете простые и краткие обозначения, чтобы компилятор создавал для вас именно эти специализации. Например, один шаблон класса `Stack` («стек») может стать основой для создания множества таких специализаций, как `Stack_of_doubles`, `Stack_of_ints`, `Stack_of_Employees`, `Stack_of_Bills` или `Stack_of_ActivationRecords`.

Для использования типа с шаблоном этот тип должен соответствовать требованиям шаблона. Например, шаблон может требовать от объектов вашего типа:

---

[1] «Google C++ Style Guide — Template Metaprogramming». Был доступен: 15 сентября 2023 г. https://google.github.io/styleguide/cppguide.html#Template_metaprogramming.

[2] C++ Core Guidelines, «T.120: Use Template Metaprogramming Only When You Really Need To». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-metameta.

[3] «C++ – Static Polymorphism». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/C%2B%2B#Static_polymorphism.

[4] Екатерина Бондаренко, «Static Polymorphism in C++», 6 мая 2019 г. Был доступен: 15 сентября 2023 г. https://medium.com/@kateolenya/static-polymorphism-in-c-9e1ac27a945b.

- возможности инициализации конструктором по умолчанию;
- поддержки семантики копирования или перемещения;
- возможности сравнения оператором < для определения порядка сортировки;
- наличия определенных функций и т. д.

При создании специализации шаблона с таким типом, который не соответствует требованиям шаблона, обычно возникают ошибки на этапе компиляции.

Создание шаблона класса Stack<T>

Давайте напишем пользовательский шаблон класса `Stack`. Понятие «стек» не зависит от элементов, которые вы помещаете в стек или удаляете из стека. Стек — это такая структура данных, в которой элемент, добавленный последним, обрабатывается первым (last-in, first-out – LIFO). Шаблоны классов стимулируют повторное использование программного обеспечения, потому что компилятор может «штамповать» по одному и тому же шаблону множество специализированных классов, как вы уже видели в разделе 13.10.1, посвященном адаптеру контейнера `stack`. Сначала (илл. 15.1) мы определим стек в обобщенном виде, а затем создадим и используем версии этого стека, специализированные для конкретных типов (илл. 15.2). Определение шаблона класса `Stack` на илл. 15.1 выглядит почти как обычное определение класса, но с несколькими ключевыми отличиями.

```
// Fig. 15.1: Stack.h
// Шаблон класса Stack
#pragma once
#include <deque>

template<typename T>
class Stack {
public:
   // Возвращаем верхний элемент стека
   const T& top() const {return stack.front();}

   // Добавляем элемент в стек
   void push(const T& pushValue) {stack.push_front(pushValue);}

   // Удаляем элемент из стека
   void pop() {stack.pop_front();}

   // Проверяем, пуст ли стек
   bool isEmpty() const {return stack.empty();}

   // Возвращаем размер стека
   size_t size() const {return stack.empty();}
private:
   std::deque<T> stack{}; // Внутреннее представление стека
};
```

Илл. 15.1 | Шаблон класса Stack

Заголовок template

Первое важное отличие шаблона — это заголовок, начинающийся с ключевого слова `template` (строка 6):

```
template<typename T>
```

В угловых скобках (< и >) после ключевого слова `typename` указан **параметр шаблона (template parameter)**, обозначающий тип данных, или сокращенно **параметр-тип (type parameter)**. Имя параметра-типа может быть любым допустимым идентификатором, но должно быть уникальным внутри определения шаблона. В данном случае мы выбрали имя `T`. В угловых скобках можно перечислить через запятую несколько параметров-типов. Каждому параметру-типу обязательно предшествует одно из взаимозаменяемых ключевых слов: `typename` или `class`. Многие программисты предпочитают `typename`, потому что тип аргумента шаблона может не быть типом класса. (Есть и другие случаи, когда необходимо использовать `typename`, а не `class`[1, 2].) Параметр-тип `T` — это **заполнитель (placeholder)**, заменяющий в коде шаблона имя типа элементов стека. Параметр-тип `T` встречается в шаблоне класса `Stack` везде, где требуется доступ к типу элементов:

- в объявлении возвращаемого типа функции (строка 10);
- в объявлении параметра функции (строка 13);
- в объявлении переменной (строка 24).

Компилятор связывает параметр-тип с типом аргумента, когда создаст специализацию шаблона класса. На этом этапе компилятор генерирует копию шаблона класса, в которой все вхождения параметра-типа заменяются на имя указанного типа. Компиляторы генерируют определения только для тех фрагментов шаблона, которые используются в вашем коде[3, 4].

Есть еще одно отличие между шаблоном класса `Stack` и другими классами, которые мы определяли в этой книге: мы не отделяем интерфейс шаблона класса от его реализации. Вы определяете шаблоны в заголовках и включаете их в файлы клиентского кода. Каждый раз, когда создается специализация шаблона с аргументами нового типа, компилятору требуется полное определение шаблона для генерации соответствующего кода. Поэтому в шаблоне класса его функции определяются в заголовке, обычно внутри определения класса, как на илл. 15.1.

---

[1] Скотт Майерс (Scott Meyers), «Item 42: Understand the Meanings of typename», «Effective C++», 3/e. Изд. Pearson Education, 2005.

[2] Джон Лакос (John Lakos), «1.3 Declarations, Definitions, and Linkage», «Large-Scale C++». Addison-Wesley, 2020. Сноски 56 и 57.

[3] Андреас Фертиг (Andreas Fertig), «Back to Basics: Templates (Part 1 of 2)», 25 сентября 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=VNJ4wiuxJM4.

[4] «13.9.2. Implicit Instantiation [temp.inst]». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/temp.inst#4.

Представление данных шаблона класса Stack<T>

В разделе 13.10.1 было показано, что класс адаптера `stack` стандартной библиотеки может использовать различные контейнеры для хранения элементов. Адаптер `stack` рассчитан на вставки и удаления только сверху, поэтому для хранения элементов стека можно использовать контейнеры `vector` и `deque`. Контейнер `vector` рассчитан на быстрые вставки и удаления в конце. Контейнер `deque` рассчитан на быстрые вставки и удаления в начале и в конце. Контейнер `deque` является контейнером по умолчанию для адаптера стандартной библиотеки `stack`[1], потому что `deque` расширяется эффективнее, чем `vector`:

- Элементы контейнера `vector` хранятся в непрерывном блоке памяти. Когда выделенный блок переполнен и вы добавляете новый элемент, контейнер выполняет ресурсозатратную операцию: выделяет в памяти непрерывный блок большего размера и копирует или перемещает старые элементы в новый блок.
- Контейнер `deque` обычно реализуется в виде списка традиционных массивов фиксированного размера — с добавлением новых массивов по мере необходимости. Существующие элементы не копируются и не перемещаются, когда новые элементы добавляются в начало или в конец контейнера.

По этим причинам мы используем `deque` (строка 24) в качестве базового контейнера нашего класса `Stack`.

Функции шаблона класса Stack<T>

Определения функций шаблона класса ведут себя аналогично шаблонам функций. Перед такими определениями в теле шаблона класса не нужен заголовок `template`. Они используют параметр `T` шаблона для представления типа элементов. Шаблон класса `Stack` не определяет свои собственные конструкторы — конструктор класса по умолчанию вызывает конструктор элементов данных `deque` по умолчанию.

Шаблон класса `Stack` (илл. 15.1) предоставляет следующие функции:

- Функция `top` (строка 10), возвращающая константную ссылку на верхний элемент стека без его удаления. Вы можете перегрузить функцию `top` для возврата неконстантной ссылки.
- Функция `push` (строка 13), добавляющая верхний элемент стека.
- Функция `pop` (строка 16), удаляющая верхний элемент стека.
- Функция `isEmpty` (строка 19) возвращающая `true`, если стек пуст, или `false` в противном случае.
- Функция `size` (строка 22), возвращающая количество элементов в стеке.

Каждая из этих функций вызывает для выполнения своей задачи соответствующую функцию контейнера `deque` — это называется **делегированием (delegation)**.

[1] «std::stack». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/container/stack.

Тест-драйв шаблона класса Stack<T>

На илл. 15.2 демонстрируется работа с шаблоном класса `Stack`. В строке 8 создается `doubleStack`. Эта переменная объявлена с типом `Stack<double>`. Компилятор связывает тип `double` с параметром-типом `T` в шаблоне класса и создает исходный код класса `Stack` с элементами `double`, хранящего элементы в `deque<double>`. В строках 15–19 вызывается функция `push` (строка 16) для добавления в `doubleStack` значений `1.1`, `2.2`, `3.3`, `4.4` и `5.5`. Далее в строках 24–27 вызываются функции `isEmpty`, `top` и `pop` в цикле `while` для удаления элементов из стека. Выходные данные показывают, что значения действительно удаляются в порядке, обратном порядку добавления: «последним пришел, первым ушел» (LIFO). Когда `doubleStack` остается пустым, цикл завершается.

```
1 // fig15_02.cpp
2 // Тест-драйв шаблона класса Stack
3 #include <iostream>
4 #include "Stack.h" // Определение шаблона класса Stack
5 using namespace std;
6
7 int main() {
8    Stack<double> doubleStack{}; // Создаем стек для чисел double
9    constexpr size_t doubleStackSize{5}; // Размер стека
10   double doubleValue{1.1}; // Первое число для добавления в стек
11
12   cout << "Pushing elements onto doubleStack\n";
13
14   // Добавляем 5 чисел double в doubleStack
15   for (size_t i{0}; i < doubleStackSize; ++i) {
16      doubleStack.push(doubleValue);
17      cout << doubleValue << ' ';
18      doubleValue += 1.1;
19 }
20
21   cout << "\n\nPopping elements from doubleStack\n";
22
23   // Удаляем элементы из doubleStack
24   while (!doubleStack.isEmpty()) { // Выполняем цикл, пока стек не пуст
25      cout << doubleStack.top() << ' '; // Отображаем верхний элемент
26      doubleStack.pop(); // Удаляем верхний элемент
27   }
28
29   cout << "\nStack is empty, cannot pop.\n";
30
31   Stack<int> intStack{}; // Создаем стек для чисел int
32   constexpr size_t intStackSize{10}; // Размер стека
33   int intValue{1}; // Первое число для добавления в стек
34
35   cout << "\nPushing elements onto intStack\n";
36
37   // Добавляем 10 чисел int в intStack
```

Окончание ⇨

```
38    for (size_t i{0}; i < intStackSize; ++i) {
39       intStack.push(intValue);
40       cout << intValue++ << ' ';
41    }
42
43    cout << "\n\nPopping elements from intStack\n";
44
45    // Удаляем элементы из intStack
46    while (!intStack.isEmpty()) { // Выполняем цикл, пока стек не пуст
47       cout << intStack.top() << ' '; // Отображаем верхний элемент
48       intStack.pop(); // Удаляем верхний элемент
49    }
50
51    cout << "\nStack is empty, cannot pop.\n";
52 }
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty, cannot pop.

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty, cannot pop.
```

Илл. 15.2 | Тест-драйв шаблона класса Stack

В строке 31 создается специализация `intStack`, объявленная как `Stack<int>`. В строках 38–41 несколько раз вызывается функция `push` (строка 39) для добавления значений в `intStack`, а в строках 46–49 несколько раз вызываются функции `isEmpty`, `top` и `pop` для удаления значений до тех пор, пока `intStack` не станет пустым. И снова выходные данные подтверждают порядок добавления и удаления элементов: «последним пришел, первым ушел» (LIFO).

Хотя компилятор не показывает код, сгенерированный для `Stack<double>` и `Stack<int>`, вы можете увидеть пример сгенерированного кода на этом веб-сайте:

https://cppinsights.io

Этот сайт показывает специализации шаблонов, созданные компилятором Clang C++[1].

[1] Этот сайт требует, чтобы весь код был вставлен в панель кода. Для тест-драйва нашего примера удалите #include stack.h в main и вставьте код шаблона класса Stack выше main. Кроме того, удалите директиву #pragma once.

Определение функций шаблона класса вне шаблона класса

Функции шаблона класса можно определять вне определения шаблона класса. В этом случае каждое определение функции должно начинаться с такого же заголовка `template`, как в шаблоне класса. Кроме того, вы должны квалифицировать каждую функцию, добавив перед ее именем имя класса и оператор разрешения области видимости. Например, вы можете определить функцию `pop` вне определения шаблона класса таким образом:

```
template<typename T>
void Stack<T>::pop() {stack.pop_front();}
```

Обозначение `Stack<T>::` указывает, что функция `pop` находится в области видимости шаблона класса `Stack<T>`. Некоторые функции шаблонов классов стандартной библиотеки определены в теле шаблона, а другие — вне тела шаблона.

15.3. Расширения функциональности шаблонов, введенные в C++20

В дополнение к концептам в C++20 добавлены сокращенные шаблоны функций и шаблоны лямбда-выражений.

15.3.1. Сокращенные шаблоны функций (C++20)

Используя традиционный синтаксис шаблона функции, мы можем определить шаблон функции `printContainer` с заголовком `template` таким образом:

```
template <typename T>
void printContainer(const T& items) {
   for (const auto& item : items) {
      std::cout << item << " ";
   }
}
```

Этот шаблон функции получает ссылку на контейнер (`items`) и отображает элементы в цикле `for` для диапазона значений. **Сокращенные шаблоны функций (abbreviated function templates)**, введенные в C++20 (илл. 15.3), позволяют определить шаблон функции без заголовка `template` (строки 10–14), используя ключевое слово `auto` в качестве параметра-типа (строка 10).

```
 1 // fig15_03.cpp
 2 // Сокращенный шаблон функции
 3 #include <array>
 4 #include <iostream>
 5 #include <string>
 6 #include <vector>
 7
 8 // Сокращенный шаблон функции printContainer отображает
 9 // элементы контейнера, разделяя их запятыми
10 void printContainer(const auto& items) {
```

Окончание ⇨

```
11    for (const auto& item : items) {
12       std::cout << item << " ";
13    }
14 }
15
16 int main() {
17    using namespace std::string_literals; // Строковые объектные литералы
18
19    std::array ints{1, 2, 3, 4, 5};
20    std::vector strings{"red"s, "green"s, "blue"s};
21
22    std::cout << "ints: ";
23    printContainer(ints);
24    std::cout << "\nstrings: ";
25    printContainer(strings);
26    std::cout << "\n";
27 }
```

```
ints: 1 2 3 4 5
strings: red green blue
```

Илл. 15.3 | Сокращенный шаблон функции

В строке 19 создается массив чисел `int`, а в строке 20 — вектор строк, инициализированный тремя строковыми объектными литералами. Компилятор использует CTAD, чтобы определить типы элементов по списку инициализаторов. При каждом вызове функции `printContainer` в строках 23 и 25 компилятор автоматически определяет параметр-тип по типу контейнера, переданного в аргументе, и генерирует соответствующую специализацию шаблона.

Иногда необходим традиционный синтаксис шаблона функции

Сокращенный шаблон функции синтаксически похож на обычное определение функции, но его не всегда целесообразно использовать. Рассмотрим первые две строки шаблона функции `maximum` из раздела 5.16, которая получает три параметра одного типа:

```
template <typename T>
T maximum(T value1, T value2, T value3) {
```

Эти строки показывают корректный способ гарантировать, что все три параметра имеют одинаковый тип.

Предположим, что мы напишем определение функции `maximum` в виде сокращенного шаблона функции:

```
auto maximum(auto value1, auto value2, auto value3) {
```

В этом случае компилятор автоматически определит тип каждого параметра по соответствующему аргументу. Таким образом, функция `maximum` сможет получать аргументы трех разных типов.

Функция printContainer и неподдерживаемые типы

При попытке создать специализацию шаблона функции `printContainer` компилятор может выдать сообщение об ошибке, если:

- мы передадим объект, несовместимый с инструкцией `for` для диапазона значений,

или

- объекты элемента контейнера не поддерживают вывод оператором `<<`.

Обычно такие сообщения непонятны, потому что в них упоминаются детали внутренней реализации функции `printContainer`, которые не должна знать вызывающая сторона. Если аргумент несовместим с реализацией функции, для программистов клиентского кода будет понятнее лаконичное сообщение об ошибке, которое просто указывает, в чем причина. В разделах 15.4 и 15.6 мы покажем, что концепты (C++20) могут использоваться для ограничения типов, передаваемых шаблону функции, и для предотвращения попыток компилятора создавать некорректные специализации шаблона. Как вы увидите, с концептами будут генерироваться такие сообщения об ошибках, которые, как правило, проще понять.

15.3.2. Шаблоны лямбда-выражений (C++20)

В C++20 лямбды могут содержать параметры шаблонов. Рассмотрим следующую лямбду с илл. 14.16, которую мы использовали для вычисления суммы квадратов элементов массива:

```
[](auto total, auto value) {return total + value * value;}
```

Компилятор автоматически определяет типы переменных `total` и `value` (возможно, разные) по их аргументам. Вы можете заставить лямбду требовать один и тот же тип для обеих переменных, используя шаблон:

```
[]<typename T>(T total, T value) {return total + value * value;}
```

Список параметров шаблона помещается между списком захвата и списком параметров лямбды. Если вы указали один и тот же параметр-тип (`T`) для объявления обоих параметров лямбды, компилятор будет требовать, чтобы оба аргумента были одного типа; в противном случае возникнет ошибка компиляции.

15.4. Концепты (C++20): первый взгляд

Концепты (кратко рассмотренные в разделе 14.2) упрощают обобщенное программирование. Эксперты C++ говорят, что «концепты — это революционный подход к написанию шаблонов»[1] и что «C++20 создает прорыв в парадигме ис-

[1] Бартломей Филипек (Bartlomiej Filipek), «C++20 Concepts — A Quick Introduction», 5 мая 2021 г. Был доступен: 15 сентября 2023 г. https://www.cppstories.com/2021/concepts-intro/.

пользования метапрограммирования»[1]. Создатель C++ Бьерн Страуструп говорит, что «концепты дополняют шаблоны C++, как и предполагалось с самого начала» и что они «значительно улучшают ваше обобщенное программирование и делают нынешние обходные пути (например, признаки классов) и низкоуровневые методы (например, перегрузку на основе `enable_if`) похожими на утомительное программирование на ассемблере»[2].

Как вы увидите, концепты явно ограничивают аргументы для параметров шаблона. Вы будете использовать условия и выражения `requires` для ограничений, которые могут:

- тестировать атрибуты типов (например, «является ли этот тип целочисленным?»);
- проверять поддержку различных операций (например, «поддерживает ли этот тип операции сравнения?»).

Стандарт C++ предоставляет вам 74 предопределенных концепта (см. раздел 15.4.3). В дополнение к ним вы можете создавать собственные концепты. Каждый из них определяет требования к типу или взаимосвязь между типами[3]. Концепт можно применить к любому параметру любого шаблона и к любому использованию `auto`[4]. Мы обсудим концепты более подробно в разделе 15.6, а сейчас сосредоточимся на их применении для ограничения параметров-типов шаблона функции.

Предпосылки и цели концептов

Страуструп говорит: «Концепты допускают перегрузки и устраняют необходимость в больших объемах бессистемного метапрограммирования и в значительной части автоматически генерируемого кода, таким образом значительно упрощая метапрограммирование, а также обобщенное программирование»[5].

Традиционно требования шаблонов были неявными, основанными на том, как шаблоны используют свои аргументы в выражениях с операторами, в вызовах функций и т. д.[6] Это видно в сокращенном шаблоне функции `printContainer` на

---

[1] Инбал Леви (Inbal Levi), «Exploration of C++20 Meta Programming», 29 сентября 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=XgrjybKaIV8.

[2] Бьерн Страуструп (Bjarne Stroustrup), «Concepts: The Future of Generic Programming – 3.1 Specifying Template Interfaces», 31 января 2017 г. Был доступен: 15 сентября 2023 г. http://wg21.link/p0557r0.

[3] Бьерн Страуструп (Bjarne Stroustrup), «Concepts: The Future of Generic Programming – 3.1 Specifying Template Interfaces», 31 января 2017 г. Был доступен: 15 сентября 2023 г. http://wg21.link/p0557r0.

[4] «Placeholder Type Specifiers». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/language/auto.

[5] Бьерн Страуструп, «Concepts: The Future of Generic Programming – 3 Using Concepts», 31 января 2017 г. Был доступен: 15 сентября 2023 г. http://wg21.link/p0557r0.

[6] Хендрик Нимейер (Hendrik Niemeyer), «An Introduction to C++20's Concepts», 25 июля 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=N_kPd2OK1L8.

илл. 15.3. В определении этой функции не указано, что аргумент должен быть итерируемым в инструкции for для диапазона значений, а тип элемента должен поддерживать оператор << для вывода. Конечно, подобные требования должны документироваться в комментариях к программе, но компилятор не умеет читать комментарии. Чтобы узнать о несовместимости типа с шаблоном, компилятор должен попытаться создать специализацию шаблона и «убедиться», что тип не поддерживает неявные требования шаблона. Это приводит ко многим сообщениям об ошибках компиляции, часто загадочным для программиста.

Концепты явно определяют требования к шаблону в коде. Они позволяют компилятору узнать о несовместимости типа с шаблоном заранее, перед созданием специализации шаблона. Это снижает количество ошибок, конкретизирует сообщения об ошибках и повышает быстродействие во время компиляции[1].

Концепты также позволяют перегружать шаблоны функций с одной и той же сигнатурой на основе требований каждого шаблона[2]. Для примера, мы можем перегрузить два шаблона функции с одинаковой сигнатурой:

- один шаблон будет поддерживать контейнеры с итераторами ввода;
- другой шаблон будет оптимизирован для контейнеров с итераторами произвольного доступа.

15.4.1. Неограниченный шаблон функции multiply

Когда вы вызываете функцию, компилятор использует правила разрешения перегрузки[3] и метод **поиска, зависящего от аргументов (argument-dependent lookup**, или сокращенно **ADL)**[4, 5, 6, 7], чтобы найти все определения функции, которые могут соответствовать этому вызову, — они называются **набором перегрузок (overload set)**. Этот процесс часто включает создание шаблона функции на основе типов аргументов вызова функции. Затем компилятор выбирает наилучшее соответствие из набора перегрузок. Неограниченный шаблон функции не содержит никаких явных требований. Поэтому компилятор просто

---

[1] Количество и содержание сообщений об ошибках по-прежнему значительно различается в зависимости от компилятора.

[2] Это позволяет перегрузить с определенными ограничениями даже конструктор класса без аргументов.

[3] «Overload Resolution». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/language/overload_resolution.

[4] Стандарт C++, «6.5.4. Argument-Dependent Name Lookup [basic.lookup.argdep]». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/basic.lookup.argdep#:lookup,argument-dependent.

[5] «Argument-Dependent Lookup». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/language/adl.

[6] Инбал Леви (Inbal Levi), «Exploration of C++20 Metaprogramming», 29 сентября 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=XgrjybKaIV8.

[7] Артур О'Двайр (Arthur O'Dwyer), «What Is ADL?», 26 апреля 2019 г. Был доступен: 15 сентября 2023 г. https://quuxplusone.github.io/blog/2019/04/26/what-is-adl/.

подставляет типы аргументов из вызова в объявление шаблона функции, проверяя, есть ли между ними соответствие. Если оно есть, компилятор проверяет, является ли оно наилучшим. Только в случае наилучшего соответствия между типами аргументов и шаблоном функции компилятор создает специализацию шаблона и выясняет, поддерживают ли эти типы операции, выполняемые в теле шаблона.

Рассмотрим неограниченный шаблон функции `multiply` (строки 5–6 на илл. 15.4), которая принимает два числа одного типа (`T`) и возвращает их произведение.

```
 1 // fig15_04.cpp
 2 // Простой неограниченный шаблон функции multiply
 3 #include <iostream>
 4
 5 template<typename T>
 6 T multiply(T first, T second) {return first * second;}
 7
 8 int main() {
 9    std::cout << "Product of 5 and 3: " << multiply(5, 3)
10       << "\nProduct of 7.25 and 2.0: " << multiply(7.25, 2.0) << "\n";
11 }
```

```
Product of 5 and 3: 15
Product of 7.25 and 2.0: 14.5
```

Илл. 15.4 | Простой неограниченный шаблон функции multiply

Этот шаблон имеет неявные требования, которые вы можете понять из кода:

- Параметры-типы (строка 6) не являются указателями или ссылками, поэтому аргументы передаются по значению. Возвращаемый тип тоже не является указателем или ссылкой, поэтому результат возвращается по значению. Таким образом, тип аргументов должен поддерживать семантику копирования или перемещения.
- Аргументы перемножаются, поэтому тип аргументов должен поддерживать бинарный оператор умножения (`*`) — либо по умолчанию (как со встроенными типами, такими как `int` и `double`), либо через перегрузку этого оператора.

Когда компилятор создает специализацию функции `multiply` для данного типа и определяет, что аргументы несовместимы с неявными требованиями шаблона, он исключает функцию из набора перегрузок. В этом примере в строках 9 и 10 вызывается функция `multiply` с двумя значениями `int` и двумя значениями `double` соответственно. Все числовые типы в C++ поддерживают неявные требования шаблона, поэтому компилятор создает специализации функции для каждого типа. Но обратите внимание, что вы не можете вызывать функцию `multiply` с аргументами разных типов, даже если тип одного аргумента может быть неявно преобразован в тип другого аргумента. В шаблоне есть только один параметр-тип, поэтому аргументы должны быть одного типа.

Несовместимые типы в шаблоне функции multiply

Что произойдет, если в функцию `multiply` передать аргументы, не поддерживающие неявные требования шаблона, а другое объявление функции, которое лучше соответствует ее вызову, отсутствует? Компилятор будет генерировать сообщения об ошибках. Например, следующий код пытается вычислить произведение двух строк[1]:

```
std::string s1{"hi"};
std::string s2{"bye"};
auto result{multiply(s1, s2)}; // Класс string не поддерживает оператор *
```

Класс `string` не поддерживает оператор `*`. Если мы добавим эти строки в основную часть программы на илл. 15.4 и перекомпилируем ее, мы получим сообщения об ошибках (мы добавили пустые строки для удобства чтения). В компиляторе Clang:

```
fig15_04.cpp:6:45: error: invalid operands to binary expression
('std::basic_string<char>' and 'std::basic_string<char>')
T multiply(T first, T second) {return first * second;}
                                      ~~~~~ ^ ~~~~~~

fig15_04.cpp:14:16: note: in instantiation of function template
specialization 'multiply<std::basic_string<char>>' requested here
   auto result{multiply(s1, s2)}; // string does not have * operator
               ^
1 error generated.
```

В компиляторе GNU g++:

```
fig15_04.cpp: In instantiation of 'T multiply(T, T) [with T =
std::__cxx11::basic_string<char>]':

fig15_04.cpp:14:24:   required from here

fig15_04.cpp:6:45: error: no match for 'operator*' (operand types are
'std::__cxx11::basic_string<char>' and 'std::__cxx11::basic_string<char>')
    6 | T multiply(T first, T second) {return first * second;}
      |                                      ~~~~~^~~~~~~~
```

В компиляторе Visual C++:

```
1>fig15_04.cpp
1>c:\pauldeitel\Documents\examples\ch15\fig15_04.cpp(6,45): error C2676: binary
'*': 'T' does not define this operator or a conversion to a type acceptable to
the predefined operator
1>        with
```

Окончание ⇨

[1] Для вашего удобства этот фрагмент приведен в файле исходного кода в виде комментариев в конце функции main.

```
1>        [
1>            T=std::string
1>        ]
1>c:\pauldeitel\Documents\examples\ch15\fig15_04.cpp(14): message : see
reference to function template instantiation 'T multiply<std::string>(T,T)'
being compiled
1>        with
1>        [
1>            T=std::string
1>        ]
1>Done building project "concurrencpp_test.vcxproj" -- FAILED.
```

Это относительно короткие и понятные сообщения, но так бывает редко. Более сложные шаблоны, как правило, приводят к ужасно длинным сообщениям об ошибках. Например, передача неправильных типов итераторов алгоритму стандартной библиотеки, который не ограничен концептами, может привести к сотням строк сообщений об ошибках. Для примера мы попытались скомпилировать программу, содержащую всего две простые инструкции сортировки контейнера `std::list`:

```
std::list integers{10, 2, 33, 4, 7, 1, 80};
std::sort(integers.begin(), integers.end());
```

Контейнер `std::list` поддерживает двунаправленные итераторы, но алгоритму `std::sort` нужны итераторы произвольного доступа. Один из наших компиляторов сгенерировал для этого простого вызова алгоритма `std::sort` более 1000 строк сообщений об ошибках! Зато алгоритм `std::ranges::sort`, ограниченный концептами, выдал гораздо меньше сообщений и пояснил в них, что алгоритму нужны итераторы произвольного доступа.

15.4.2. Ограниченный шаблон функции с условием requires (C++20)

Концепты позволяют вам указывать ограничения. Каждое из них представляет собой предикатное выражение для компилятора, принимающее значение `true` или `false`[1]. Компилятор использует ограничения для проверки требований к типу перед созданием специализаций шаблонов. В C++ Core Guidelines рекомендуется:

- указывать концепты для каждого параметра шаблона[2];
- по возможности использовать стандартные предопределенные концепты[3].

---

[1] Стандарт C++, «13.5. Template Constraints». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/temp.constr.

[2] C++ Core Guidelines, «T.10: Specify Concepts for All Template Arguments». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-concepts.

[3] C++ Core Guidelines, «T.11: Whenever Possible Use Standard Concepts». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-std-concepts.

Вы можете применить концепты к параметру шаблона функции, чтобы явно указать требования к типу ее аргумента. Если тип аргумента соответствует этим требованиям, компилятор создает специализацию шаблона; в противном случае шаблон игнорируется. Компилятор не находит возможности вызова функции и генерирует такие сообщения об ошибках, которые обычно короче (причем намного короче), понятнее и точнее. Это преимущество концептов перед другими методами, которые они частично или полностью заменяют.

Условие requires

На илл. 15.5 в строке 8 используется введенное в C++20 **условие** `requires` (**requires clause**), ограничивающее параметр `T` шаблона `multiply`. После ключевого слова `requires` следует **ограничивающее выражение (constraint expression)**, состоящее из одного или нескольких вычисляемых при компиляции логических выражений в комбинации с логическими операторами `&&` и `||`.

```
1 // fig15_05.cpp
2 // Ограниченный шаблон функции multiply, разрешающий
3 // только целые числа и числа с плавающей точкой
4 #include <concepts>
5 #include <iostream>
6
7 template<typename T>
8    requires std::integral<T> || std::floating_point<T>
9 T multiply(T first, T second) {return first * second;}
10
11 int main() {
12    std::cout << "Product of 5 and 3: " << multiply(5, 3)
13       << "\nProduct of 7.25 and 2.0: " << multiply(7.25, 2.0) << "\n";
14
15    std::string s1{"hi"};
16    std::string s2{"bye"};
17    auto result{multiply(s1, s2)};
18 }
```

```
fig15_05.cpp: In function 'int main()':
fig15_05.cpp:17:24: error: no matching function for call to
'multiply(std::string&, std::string&)'
   17 |    auto result{multiply(s1, s2)};
      |                 ~~~~~~~~^~~~~~~~

fig15_05.cpp:9:3: note: candidate: 'template<class T> requires (integral<T>)
|| (floating_point<T>) T multiply(T, T)'
    9 | T multiply(T first, T second) {return first * second;}
      |   ^~~~~~~~

fig15_05.cpp:9:3: note: template argument deduction/substitution failed:
fig15_05.cpp:9:3: note: constraints not satisfied
fig15_05.cpp: In substitution of 'template<class T> requires (integral<T>)
|| (floating_point<T>) T multiply(T, T) [with T = std::__cxx11::ba-
sic_string<char>]':
```

Окончание ⇨

```
fig15_05.cpp:17:24: required from here
fig15_05.cpp:9:3: required by the constraints of 'template<class T>
requires (integral<T>) || (floating_point<T>) T multiply(T, T)'

fig15_05.cpp:8:30: note: no operand of the disjunction is satisfied
    8 |    requires std::integral<T> || std::floating_point<T>
      |              ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Илл. 15.5 | Ограниченный шаблон функции multiply, разрешающий только целые числа и числа с плавающей точкой (в выходных данных — сообщения об ошибках компилятора g++)

В строке 8 указано, что типы аргументов функции `multiply` должны соответствовать хотя бы одному из следующих условий, которые являются ограничениями для параметра-типа `T`:

- `std::integral<T>` означает, что тип `T` может быть любым целочисленным типом данных.
- `std::floating_point<T>` означает, что тип `T` может быть любым типом данных с плавающей точкой.

Все целочисленные типы и типы с плавающей точкой поддерживают арифметические операторы, поэтому мы можем быть уверены, что все значения этих типов будут работать с оператором `*` в строке 9. Если ни одно из перечисленных ограничений не выполняется, то типы аргументов несовместимы с шаблоном функции `multiply`, поэтому компилятор не будет создавать специализацию шаблона. Если никакие другие функции не соответствуют вызову, компилятор выдаст сообщения об ошибках.

Далее вы увидите, что некоторые концепты могут устанавливать много индивидуальных ограничений. Показанные в этом примере концепты из заголовка `<concepts>` — это только 2 из 74 предопределенных концептов, введенных в C++20[1]. В разделе 15.4.3 перечислены все предопределенные концепты, их категории и заголовки.

Дизъюнкция и конъюнкция

Логический оператор ИЛИ (`||`) в строке 8 образует дизъюнкцию. Один или оба операнда должны иметь значение `true`, чтобы компилятор мог создать специализацию шаблона. Если оба операнда имеют значение `false`, компилятор игнорирует шаблон, считая его непригодным для вызова функции `multiply`. Функция `multiply` определяет только один параметр-тип, поэтому оба операнда должны иметь одинаковый тип. Таким образом, только одно условие в строке 8 должно быть истинным для каждого вызова `multiply` в этом примере.

[1] Стандарт C++, «Index of Library Concepts». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/conceptindex.

Вы можете сформировать конъюнкцию с логическим оператором И (`&&`), указывая, что для создания специализации шаблона оба операнда должны быть истинными. Дизъюнкции и конъюнкции используют вычисления по короткой схеме (раздел 4.11.3).

Вызов функции multiply с аргументами, соответствующими ограничениям

В строках 12 и 13 вызывается функция `multiply` с двумя значениями `int` и двумя значениями `double` соответственно. Когда компилятор ищет определение функции, соответствующее каждому из этих вызовов, он встречает только шаблон функции `multiply` в строках 7–9. Компилятор проверяет типы аргументов, определяя, соответствуют ли они хотя бы одному из условий `requires` в строке 8. Оба аргумента каждого вызова соответствуют одному из требований дизъюнкции, поэтому компилятор создает специализацию шаблона для типа `int` в строке 12 и для типа `double` в строке 13. Напомним, что шаблон функции `multiply` имеет только один параметр-тип, поэтому оба аргумента должны быть одного типа, иначе компилятор не сможет определить, какой тип использовать для создания специализации шаблона, и выдаст ошибку.

Вызов функции multiply с аргументами, не соответствующими ограничениям

В строке 17 вызывается функция `multiply` с двумя аргументами `string`. Когда компилятор ищет определение функции, соответствующее вызову, он встречает только шаблон функции `multiply` и проверяет типы аргументов, определяя, соответствуют ли они хотя бы одному из условий `requires`, перечисленных в строке 8. Тип `string` не соответствует ни одному из условий. Кроме того, никакие другие функции `multiply` не могут принять два аргумента типа `string`, поэтому компилятор выдает сообщения об ошибках. На илл. 15.5 показаны сообщения об ошибках компилятора g++. Мы выделили несколько сообщений жирным шрифтом и добавили пустые строки для удобства чтения. Обратите внимание на последние несколько строк, где компилятор сообщает: «no operand of the disjunction is satisfied» и ссылается на условие `requires` в строке 8. В этом примере самым простым, по сравнению с двумя другими нашими компиляторами, оказалось сообщение Visual C++:

```
1>c:\pauldeitel\Documents\examples\ch15\fig15_05.cpp(17,16): error
C2672: 'multiply': no matching overloaded function found
1>c:\pauldeitel\Documents\examples\ch15\fig15_05.cpp(17,31): error
C7602: 'multiply': the associated constraints are not satisfied
1>c:\pauldeitel\Documents\examples\ch15\fig15_05.cpp(9): message :
see declaration of 'multiply'
```

15.4.3. Концепты, предопределенные в C++20

В разделе «Index of library concepts» стандарта C++[1] перечислены в алфавитном порядке концепты, определенные в этом стандарте. В следующей таблице мы

[1] Стандарт C++, «Index of Library Concepts». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/conceptindex.

перечислили все концепты, определенные в заголовках <concepts>, <iterator>, <ranges>, <compare> и <random>. Мы разделили 31 концепт из заголовка <concepts> на подкатегории: «основные концепты языка», «концепты сравнения», «концепты объектов» и «концепты вызовов». Многие концепты имеют интуитивно понятные названия. Подробную информацию о каждом из них вы найдете на странице соответствующего заголовка на сайте cppreference.com.

74 предопределенных концепта, введенных в C++20		
Заголовок <concepts>, основные концепты языка		
assignable_from	default_initializable	same_as
common_reference_with	derived_from	signed_integral
common_with	destructible	swappable
constructible_from	floating_point	swappable_with
convertible_to	integral	unsigned_integral
copy_constructible	move_constructible	
Заголовок <concepts>, концепты сравнения		
equality_comparable	totally_ordered	totally_ordered_with
equality_comparable_with		
Заголовок <concepts>, концепты объектов		
copyable	regular	semiregular
movable		
Заголовок <concepts>, концепты вызовов		
equivalence_relation	predicate	relation
invocable	regular_invocable	strict_weak_order
Заголовок <iterator>		
bidirectional_iterator	indirectly_copyable_storable	input_or_output_iterator
contiguous_iterator	indirectly_movable	mergeable
forward_iterator	indirectly_movable_storable	output_iterator
incrementable	indirectly_readable	permutable
indirect_binary_predicate	indirectly_regular_unary_invocable	random_access_iterator
indirect_equivalence_relation		sentinel_for
indirect_strict_weak_order	indirectly_swappable	sized_sentinel_for
indirect_unary_predicate	indirectly_unary_invocable	sortable

74 предопределенных концепта, введенных в C++20		
`indirectly_comparable`	`indirectly_writable`	`weakly_incrementable`
`indirectly_copyable`	`input_iterator`	
Заголовок `<ranges>`		
`bidirectional_range`	`forward_range`	`random_access_range`
`borrowed_range`	`input_range`	`sized_range`
`common_range`	`output_range`	`view_range`
`contiguous_range`	`range`	`viewable_range`
Заголовок `<compare>`		
`three_way_comparable`	`three_way_comparable_with`	
Заголовок `<random>`		
`uniform_random_bit_generator`		

15.5. Признаки типов

В C++11 был введен заголовок `<type_traits>`[1] для:

- тестирования типов на наличие различных признаков во время компиляции;
- генерации кода шаблона на основе этих признаков.

Например, вы могли бы проверить, что тип:

- является фундаментальным, например `int` (это признак типов `std::is_fundamental`);
- является классом (это признак типов `std::is_class`).

Это позволяет использовать разные варианты кода шаблона для обработки конкретных случаев. В каждой последующей версии C++ добавлялось больше признаков, а некоторые из них объявлялись устаревшими или удалялись. В последней версии C++20 добавлено 10 признаков типов.

Работа с признаками типов до C++20

До появления концептов признаки типов использовались в неограниченных определениях шаблонов для проверки соответствия типов аргументов требованиям шаблона. Как и в случае с концептами, такие проверки осуществляются во время компиляции, но в процессе создания специализации шаблона, что часто приводит к большому количеству загадочных сообщений об ошибках. В отличие от признаков типов, концепты проверяются компилятором до создания специализаций

[1] «Standard Library Header <type_traits>». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/header/type_traits.

шаблонов, что обычно приводит к меньшему количеству более детальных сообщений об ошибках, чем при работе с признаками типов в неограниченных шаблонах.

Предопределенные в C++20 концепты часто используют признаки типов

В C++20 многие концепты реализованы на основе признаков типов. Например:

- концепт `std::integral` реализован на основе признака типов `std::is_integral`;
- концепт `std::floating_point` реализован на основе признака типов `std::is_floating_point`;
- концепт `std::destructible` реализован на основе признака типов `std::is_nothrow_destructible`.

Демонстрация работы с признаками типов

На илл. 15.6 показан пример работы с признаками типов, соответствующими концептам на илл. 15.5.

```
// fig15_06.cpp
// Проверка по признакам типов: являются ли типы
// целочисленными, с плавающей точкой или арифметическими?
#include <fmt/format.h>
#include <iostream>
#include <string>
#include <type_traits>

int main() {
   std::cout << fmt::format("{}\n{}{}\n{}{}\n{}{}\n{}{}\n{}{}\n\n",
      "CHECK WITH TYPE TRAITS WHETHER TYPES ARE INTEGRAL",
      "std::is_integral<int>::value: ", std::is_integral<int>::value,
      "std::is_integral_v<int>: ", std::is_integral_v<int>,
      "std::is_integral_v<long>: ", std::is_integral_v<long>,
      "std::is_integral_v<float>: ", std::is_integral_v<float>,
      "std::is_integral_v<std::string>: ",
      std::is_integral_v<std::string>);

   std::cout << fmt::format("{}\n{}{}\n{}{}\n{}{}\n{}{}\n{}{}\n\n",
      "CHECK WITH TYPE TRAITS WHETHER TYPES ARE FLOATING POINT",
      "std::is_floating_point<float>::value: ",
      std::is_floating_point<float>::value,
      "std::is_floating_point_v<float>: ",
      std::is_floating_point_v<float>,
      "std::is_floating_point_v<double>: ",
      std::is_floating_point_v<double>,
      "std::is_floating_point_v<int>: ",
      std::is_floating_point_v<int>,
      "std::is_floating_point_v<std::string>: ",
      std::is_floating_point_v<std::string>);

```

```
32    std::cout << fmt::format("{}\n{}{}\n{}{}\n{}{}\n{}{}\n",
33       "CHECK WITH TYPE TRAITS WHETHER TYPES CAN BE USED IN ARITHMETIC",
34       "std::is_arithmetic<int>::value: ", std::is_arithmetic<int>::value,
35       "std::is_arithmetic_v<int>: ", std::is_arithmetic_v<int>,
36       "std::is_arithmetic_v<double>: ", std::is_arithmetic_v<double>,
37       "std::is_arithmetic_v<std::string>: ",
38       std::is_arithmetic_v<std::string>);
39 }
```

```
CHECK WITH TYPE TRAITS WHETHER TYPES ARE INTEGRAL
std::is_integral<int>::value: true
std::is_integral_v<int>: true
std::is_integral_v<long>: true
std::is_integral_v<float>: false
std::is_integral_v<std::string>: false

CHECK WITH TYPE TRAITS WHETHER TYPES ARE FLOATING POINT
std::is_floating_point<float>::value: true
std::is_floating_point_v<float>: true
std::is_floating_point_v<double>: true
std::is_floating_point_v<int>: false
std::is_floating_point_v<std::string>: false

CHECK WITH TYPE TRAITS WHETHER TYPES CAN BE USED IN ARITHMETIC
std::is_arithmetic<int>::value: true
std::is_arithmetic_v<int>: true
std::is_arithmetic_v<double>: true
std::is_arithmetic_v<std::string>: false
```

Илл. 15.6 | Проверка по признакам типов: являются ли типы целочисленными, с плавающей точкой или арифметическими?

Каждый класс признака типа из этого примера имеет элемент `static constexpr bool` с именем `value`. Выражение в строке 12:

```
std::is_integral<int>::value
```

вычисляется на этапе компиляции как `true`, если тип в угловых скобках (`int`) является целочисленным; в противном случае выражение вычисляется как `false`.

Обозначение `::value` обычно используется с классами признаков типов для доступа к значениям `true` или `false`, поэтому в C++17 были добавлены удобные сокращения для значений признаков типов. Они определяются как шаблоны переменных (начиная с C++14) с именами, заканчивающимися на `_v`[1]. Точно так же как шаблоны функций определяют группы связанных функций, а шаблоны классов определяют группы связанных классов, шаблоны переменных определяют группы связанных переменных. Вы специализируете шаблон переменной, чтобы ее использовать. Шаблон переменной, соответствующий выражению:

```
std::is_integral<int>::value
```

[1] C++ Core Guidelines, «T.142: Use Template Variables to Simplify Notation». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-var.

определяется в стандарте C++[1] так:

```
template<class T>
inline constexpr bool is_integral_v = is_integral<T>::value;
```

Поэтому выражение `std::is_integral_v<int>` эквивалентно выражению:

```
std::is_integral<int>::value
```

В строках 12, 22 и 34 проверяются признаки типов и отображаются значения их элементов `value`, обозначающие результаты.. В других проверках используются более удобные шаблоны переменных с именами, заканчивающимися на `_v`:

- в строках 13, 14, 15 и 17 используется `std::is_integral_v` для проверки различных типов на соответствие целочисленным типам;
- в строках 24, 26, 28 и 30 используется `std::is_floating_point_v` для проверки различных типов на соответствие типам с плавающей точкой;
- в строках 35, 36 и 38 используется `std::is_arithmetic_v` для проверки различных типов на соответствие арифметическим типам, которые можно использовать в арифметических выражениях.

В следующей таблице перечислены признаки типов из заголовка `<type_traits>` и поддерживаемые функции по категориям. Новые функции, добавленные в C++14 (два новых элемента), C++17 (14 новых элементов) и C++20 (10 новых элементов), обозначены соответствующими номерами версий. Более подробную информацию о признаках типов вы найдете здесь:

https://en.cppreference.com/w/cpp/header/type_traits

В нашей таблице признаки типов перечислены в тех же категориях и в том же порядке, что на сайте cppreference.com.

Признаки типов по категориям		
Вспомогательные классы		
`bool_constant` (17)	`false_type`	`true_type`
`integral_constant`		
Основные категории типов		
`is_void`	`is_enum`	`is_lvalue_reference`
`is_null_pointer` (14)	`is_union`	`is_rvalue_reference`
`is_integral`	`is_class`	`is_member_object_pointer`
`is_floating_point`	`is_function`	`is_member_function_pointer`
`is_array`	`is_pointer`	

[1] Стандарт C++, «20.15.3. Header <type_traits> Synopsis». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/meta#type.synop.

Признаки типов по категориям		
Составные категории типов		
is_fundamental	is_object	is_reference
is_arithmetic	is_compound	is_member_pointer
is_scalar		
Свойства типов		
has_unique_object_representations (17)	is_standard_layout	is_signed
	is_empty	is_unsigned
is_const	is_polymorphic	is_bounded_array (20)
is_volatile	is_abstract	is_unbounded_array (20)
is_trivial	is_final (14)	is_scoped_enum (23)
is_trivially_copyable	is_aggregate (17)	
Поддерживаемые операции		
is_constructible	is_move_constructible	is_trivially_move_assignable
is_trivially_constructible	is_trivially_move_constructible	is_nothrow_move_assignable
is_nothrow_constructible		is_destructible
is_default_constructible	is_nothrow_move_constructible	is_trivially_destructible
is_trivially_default_constructible		is_nothrow_destructible
	is_assignable	has_virtual_destructor
is_nothrow_default_constructible	is_trivially_assignable	is_swappable_with (17)
	is_nothrow_assignable	is_swappable (17)
is_copy_constructible	is_copy_assignable	is_nothrow_swappable_with (17)
is_trivially_copy_constructible	is_trivially_copy_assignable	is_nothrow_swappable (17)
	is_nothrow_copy_assignable	
is_nothrow_copy_constructible	is_move_assignable	
Запросы свойств		
alignment_of	rank	extent
Связи между типами		
is_same	is_layout_compatible (20)	is_invocable_r (17)

Признаки типов по категориям		
is_base_of	is_pointer_interconvertible_base_of (20)	is_nothrow_invocable (17)
is_convertible	is_invocable (17)	is_nothrow_invocable_r (17)
is_nothrow_convertible (20)		
Квалификаторы const и volatile		
remove_cv	remove_volatile	add_const
remove_const	add_cv	add_volatile
Ссылки		
remove_reference	add_lvalue_reference	add_rvalue_reference
Указатели		
remove_pointer	add_pointer	
Модификаторы знака		
make_signed	make_unsigned	
Массивы		
remove_extent	remove_all_extents	
Различные преобразования		
aligned_storage	conditional	underlying_type
aligned_union	common_type	invoke_result (17)
decay	common_reference (20)	void_t (17)
remove_cvref (20)	basic_common_reference (20)	type_identity (20)
enable_if		
Операции с признаками		
conjunction (17)	disjunction (17)	negation (17)
Функции		
is_pointer_interconvertible_with_class (20)		is_corresponding_member (20)

15.6. Концепты: детальное рассмотрение

Итак, мы рассмотрели пример ограничения параметра шаблона условием `requires` и предопределенными концептами. Теперь мы создадим пользовательский концепт, объединяющий два предопределенных концепта и предоставляющий дополнительные возможности.

15.6.1. Создание пользовательского концепта

Предопределенные концепты C++20 часто объединяют несколько ограничений, в том числе иногда и ограничения из других предопределенных концептов, эффективно создавая то, что некоторые разработчики называют **категориями типов (type category)**[1]. Вы можете использовать шаблоны для определения ваших собственных, то есть пользовательских, концептов. Давайте создадим пользовательский концепт, объединяющий предопределенные концепты `std::integral` и `std::floating_point`, которыми мы в предыдущем примере (илл. 15.5) ограничивали шаблон функции `multiply`:

```
template<typename T>
concept Numeric = std::integral<T> || std::floating_point<T>;
```

Концепт начинается с заголовка `template`, за которым следуют:

- ключевое слово `concept`, введенное в C++20;
- имя концепта (`Numeric`);
- знак равенства (`=`);
- ограничивающее выражение (`std::integral<T> || std::floating_point<T>`).

Ограничивающее выражение (constraint expression) — это логическое выражение, вычисляемое на этапе компиляции и определяющее, соответствует ли аргумент шаблона заданному набору требований. В данном случае — является ли аргумент целым числом или числом с плавающей точкой. Ограничивающие выражения обычно содержат предопределенные концепты, признаки типов и, как будет показано в разделе 15.6.5, выражения `requires`, позволяющие добавлять другие требования. Заголовок `template` концепта `Numeric` содержит один параметр-тип, представляющий проверяемый тип.

Концепты с несколькими параметрами шаблона также могут регламентировать отношения между типами[2]. Например, если шаблон имеет два параметра-типа, вы можете использовать предопределенный концепт `std::same_as`, чтобы гарантировать, что типы аргументов идентичны:

```
template<typename T, typename U>
   requires std::same_as<T, U>
// Остальная часть определения шаблона
```

В разделе 15.12 мы рассмотрим вариативные шаблоны, которые могут иметь произвольное количество аргументов. Там мы будем использовать концепт `std::same_as`

---

[1] Инбал Леви (Inbal Levi), «Exploration of C++20 Metaprogramming», 29 сентября 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=XgrjybKaIV8.

[2] Бьерн Страуструп (Bjarne Stroustrup), «Concepts: The Future of Generic Programming — 3.1. Specifying Template Interfaces», 31 января 2017 г. Был доступен: 15 сентября 2023 г. http://wg21.link/p0557r0.

в вариативном шаблоне функции, требующем, чтобы все его аргументы имели одинаковый тип.

15.6.2. Использование концепта

Когда вы определили концепт, вы можете использовать его для ограничения параметра шаблона одним из четырех способов. Мы покажем первые три в этом разделе, а четвертый — в разделе 15.6.3.

Условие requires после заголовка template

Любой концепт можно включить в условие `requires` после заголовка `template`, как было показано на илл. 15.5. Вот шаблон функции `multiply` с пользовательским концептом `Numeric<T>`:

```
template<typename T>
   requires Numeric<T>
T multiply(T first, T second) {return first * second;}
```

Условие requires после сигнатуры шаблона функции

Условие `requires` можно указать после сигнатуры шаблона функции и перед телом шаблона функции:

```
template<typename T>
T multiply(T first, T second) requires Numeric<T> {
   return first * second;
}
```

Условие `requires`, указанное после сигнатуры, должно использоваться в двух случаях[1]:

- функция, определенная в теле шаблона класса, не содержит заголовка `template`, поэтому вы должны поместить условие `requires` после сигнатуры;
- чтобы использовать имена параметров шаблона функции в ограничении, вы должны расположить условие `requires` после сигнатуры, чтобы имена параметров находились в области видимости до начала оценки компилятором условия `requires`.

Концепт как тип в заголовке template

Для тех случаев, когда у вас есть только один параметр-тип, ограниченный концептом, в C++ Core Guidelines рекомендуется использовать имя концепта вместо

[1] «What Is the Difference Between the Three Ways of Applying Constraints to a Template?». Был доступен: 15 сентября 2023 г. https://stackoverflow.com/a/61875483. Примечание: в вопросе на stackoverflow.com упоминаются три способа применения ограничений к шаблону, но в ответе их четыре.

typename в заголовке template[1]. Это упрощает определение шаблона за счет отсутствия условия requires:

```
template<Numeric T>
Number multiply(T first, T second) {return first * second;}
```

Теперь каждый параметр должен соответствовать требованиям концепта Numeric. Шаблон функции по-прежнему содержит только один параметр-тип, поэтому аргументы функции должны быть одного типа, иначе возникает ошибка компиляции.

15.6.3. Концепты в сокращенных шаблонах функций

На илл. 15.3 представлены сокращенные шаблоны функций с параметрами, которые объявлены как auto, чтобы компилятор автоматически определил их типы. Везде, где допустимо использование auto в коде, вы можете добавлять имя концепта, чтобы ограничивать допустимые типы[2]. Приведем несколько примеров:

- сокращенные списки параметров шаблона функции;
- auto-типы возвращаемых значений функций;
- auto-типы локальных переменных в их определениях (определяемые по инициализаторам);
- обобщенные лямбда-выражения.

На илл. 15.7 шаблон multiply реализован как **сокращенный шаблон функции (abbreviated function template)**. В этом случае мы использовали **«auto с ограничениями» (constrained auto)** для каждого параметра, используя наш концепт Numeric, чтобы ограничить типы, которые можно передавать в качестве аргументов. Мы также использовали auto для возвращаемого типа, поэтому компилятор определит его по типу результата выражения в строке 12.

```
 1 // fig15_07.cpp
 2 // Ограниченный сокращенный шаблон функции multiply
 3 #include <concepts>
 4 #include <iostream>
 5
 6 // Концепт Numeric содержит std::integral и std::floating_point
 7 template<typename T>
 8 concept Numeric = std::integral<T> || std::floating_point<T>;
 9
10 // Сокращенный шаблон функции с ограничением автоопределения типа
11 auto multiply(Numeric auto first, Numeric auto second) {
12    return first * second;
13 }
```

Окончание ⇨

[1] C++ Core Guidelines, «T.13: Prefer the Shorthand Notation for Simple, Single-Type Argument Concepts». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-shorthand.

[2] «Placeholder Type Specifiers». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/language/auto.

```
14
15 int main() {
16    std::cout << "Product of 5 and 3: " << multiply(5, 3)
17       << "\nProduct of 7.25 and 2.0: " << multiply(7.25, 2.0)
18       << "\nProduct of 5 and 7.25: " << multiply(5, 7.25) << "\n";
19 }
```

```
Product of 5 and 3: 15
Product of 7.25 and 2.0: 14.5
Product of 5 and 7.25: 36.25
```

Илл. 15.7 | Ограниченный сокращенный шаблон функции multiply

Ключевое различие между этим сокращенным шаблоном функции и ограниченными шаблонами функций `multiply` из раздела 15.4.2 заключается в том, что компилятор рассматривает каждый параметр `auto` в строке 11 как отдельный параметр-тип шаблона. Таким образом, параметры `first` и `second` могут иметь разные типы данных, как в строке 18, которая передает в функцию два значения: `int` и `double`. Наша предыдущая версия шаблона `multiply`, ограниченная концептом, генерировала ошибки компиляции при разных типах аргументов. Код в строках 11–13 фактически эквивалентен шаблону с двумя параметрами-типами:

```
template<Numeric T1, Numeric T2>
auto multiply(T1 first, T2 second) {return first * second;}
```

Если вам нужен один и тот же тип для двух или более параметров, используйте:

- обычный шаблон функции вместо сокращенного шаблона;
- одно и то же имя параметра-типа для всех параметров функции, которые должны иметь один и тот же тип.

15.6.4. Перегрузка на основе концептов

Шаблоны функций и перегрузка тесно связаны. Когда перегруженные функции выполняют синтаксически идентичные операции с разными типами, они могут быть реализованы более компактно и удобно с помощью шаблонов функций. Затем вы можете вызывать функции с различными типами аргументов, позволяя компилятору создавать специализацию шаблона для каждого вызова функции. Все специализации имеют одно и то же имя функции, поэтому компилятор использует правила перегрузки для вызова соответствующей функции.

Процесс выбора перегруженной функции

Выбирая вызываемую функцию, компилятор просматривает функции и шаблоны функций, чтобы найти соответствующую готовую функцию или сгенерировать специализированный шаблон функции, соответствующий вызову:

- если соответствие не найдено, компилятор выдает сообщение об ошибке;
- если найдено несколько соответствий вызову функции, компилятор выбирает наилучшее из них;

- если найдено более одного наилучшего соответствия, вызов считается неоднозначным, поэтому компилятор выдает сообщение об ошибке[1].

Перегрузка шаблонов функций

Вы также можете перегружать шаблоны функций. Например, вы можете предоставить другие шаблоны функции, имеющие другие сигнатуры. Шаблон функции также может быть перегружен нешаблонной функцией с тем же именем, но с другими параметрами.

Перегрузка шаблонов функций с концептами

В C++20 вы можете использовать концепты в шаблонах функций для перегрузки на основе требований к типам[2]. Рассмотрим стандартные функции `std::distance` и `std::advance` из заголовка `<iterator>`, которые работают с контейнерами через итераторы:

- функция `std::distance` вычисляет количество элементов между двумя итераторами[3];
- функция `std::advance` перемещает итератор на n позиций от его текущей позиции[4].

Каждая из них может быть реализована в виде $O(n)$ операций:

- функция `std::distance` может использовать оператор `++` для итераций от начального до конечного итератора (но не включая конечный итератор) и для подсчета количества операций инкремента (n);
- функция `std::advance` может выполнять цикл n раз, увеличивая итератор один раз за итерацию.

Некоторые алгоритмы могут быть оптимизированы для определенных типов итераторов. И `std::distance`, и `std::advance` могут реализовать $O(1)$ операций для итераторов произвольного доступа:

- `std::distance` может использовать `operator-` для вычитания начального итератора из конечного итератора, чтобы вычислить количество элементов между ними за одну операцию;

---

[1] Процесс согласования вызовов функций компилятором очень сложен. Полная информация содержится в стандарте C++, в разделе «12.2. Overload Resolution». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/over.match.

[2] Бьерн Страуструп (Bjarne Stroustrup), «Concepts: The Future of Generic Programming — 6. Concept Overloading», 31 января 2017 г. Был доступен: 15 сентября 2023 г. http://wg21.link/p0557r0.

[3] «std::distance». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/iterator/distance.

[4] «std::advance». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/iterator/advance.

- `std::advance` может использовать `operator+` для прибавления целого числа *n* к итератору, перемещая итератор на *n* позиций за одну операцию.

На илл. 15.8 показана наша реализация перегруженных шаблонов функции `customDistance`. Каждый из них требует два итератора в качестве аргументов и вычисляет количество элементов между ними. Мы используем **перегрузку на основе концептов (concept-based overloading)**, также называемую **перегрузкой с концептами (concept overloading)**, позволяя компилятору выбирать между перегрузками, основываясь на концептах, которыми мы ограничили параметры каждого шаблона.

```
// fig15_08.cpp
// Перегрузка на основе концептов
#include <array>
#include <iostream>
#include <iterator>
#include <list>

// Вычисляем расстояние (количество элементов) между двумя итераторами ввода;
// для этого необходим последовательный перебор элементов,
// поэтому требуется O(n) операций
template <std::input_iterator Iterator>
auto customDistance(Iterator begin, Iterator end) {
   std::cout << "Called customDistance with input iterators\n";
   std::ptrdiff_t count{0};

   // Выполняем итерации от begin до end и вычисляем количество элементов
   for (auto& iter{begin}; iter != end; ++iter) {
      ++count;
   }

   return count;
}

// Вычисляем расстояние (количество элементов) между двумя итераторами
// произвольного доступа; для этого требуется O(1) операций
template <std::random_access_iterator Iterator>
auto customDistance(Iterator begin, Iterator end) {
   std::cout << "Called customDistance with random access iterators\n";
   return end - begin; // Возвращаем значение std::ptrdiff_t
}

int main() {
   std::array ints1{1, 2, 3, 4, 5}; // Предоставляет итераторы произвольного
доступа
   std::list ints2{1, 2, 3}; // Предоставляет двунаправленные итераторы

   auto result1{customDistance(ints1.begin(), ints1.end())};
   std::cout << "ints1 number of elements: " << result1 << "\n";
   auto result2{customDistance(ints2.begin(), ints2.end())};
```

```
39    std::cout << "ints2 number of elements: " << result2 << "\n";
40 }
```

```
Called customDistance with random-access iterators
ints1 number of elements: 5
Called customDistance with input iterators
ints2 number of elements: 3
```

Илл. 15.8 | Перегрузка на основе концептов

Чтобы различать шаблоны функций, мы используем предопределенные концепты `std::random_access_iterator` и `std::input_iterator` из заголовка `<iterator>`:

- функция `customDistance`, выполняющая $O(n)$ операций, в строках 11–22 требует двух аргументов, соответствующих концепту `std::input_iterator`;
- функция `customDistance`, выполняющая $O(1)$ операций, в строках 26–30 требует двух аргументов, соответствующих концепту `std::random_access_iterator`.

C++ использует тип `std::ptrdiff_t` (строка 14) для представления разницы между двумя указателями или двумя итераторами. Алгоритм `std::distance` возвращает этот тип, и мы поступаем аналогичным образом в реализациях `customDistance`.

В строках 33–34 определяются контейнеры `array` и `list`. В строке 36 вызывается функция `customDistance` для контейнера `ints1`, поддерживающего итераторы произвольного доступа. Наша функция, требующая итераторов ввода, может получать и итераторы произвольного доступа. Когда вызову функции соответствует несколько шаблонов, компилятор вызывает наиболее подходящую версию[1, 2]. Таким образом, в строке 36 вызывается $O(1)$-версия функции `customDistance` (строки 26–30). Основываясь на их определениях, компилятор знает, что `random_access_iterator` более ограничен, чем `input_iterator`. В строке 38 двунаправленные итераторы контейнера `list` не соответствуют ограничениям шаблона функции в строках 26–30, поэтому данная версия исключается из рассмотрения и вызывается более медленная $O(n)$-версия функции `customDistance`, определенная в строках 11–22.

15.6.5. Выражения requires

В C++ Core Guidelines рекомендуется использовать стандартные концепты[3]. Для пользовательских требований, которые не могут быть выражены с помощью стан-

---

[1] Хендрик Нимейер (Hendrik Niemeyer), «An Introduction to C++20's Concepts», 25 июля 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=N_kPd2OK1L8.

[2] Стандарт C++, «13.5. Template Constraints». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/temp.constr.

[3] C++ Core Guidelines, «T.11: Whenever Possible Use Standard Concepts». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-std-concepts.

дартных концептов в **условии** `requires` (`requires` **clause**), вы можете использовать **выражение** `requires` (`requires` **expression**), имеющее две формы:

```
requires {
   определения требований
}
```

или:

```
requires (список параметров) {
   определения требований, опционально использующие параметры
}
```

Список параметров похож на список параметров функции, но в нем не может быть аргументов по умолчанию. Компилятор использует параметры выражения `requires` только для проверки соответствия типов требованиям, определенным в фигурных скобках выражения. Фигурные скобки могут содержать любую комбинацию из четырех типов требований, описанных ниже: простые, типовые, составные и вложенные. Каждое требование заканчивается точкой с запятой (`;`).

Простое требование

Простое требование (**simple requirement**) проверяет, является ли выражение допустимым. Рассмотрим определение концепта `range` (заголовок `<ranges>`), который проверяет, является ли объект диапазоном C++20 с начальным итератором и конечным ограничителем[1]:

```
1 template<class T>
2 concept range =
3    requires(T& t) {
4       std::ranges::begin(t);
5       std::ranges::end(t);
6    };
```

Требования могут ссылаться на один или несколько параметров выражения и параметров шаблона из заголовка `template`. Концепт `range` определяет параметр `t` с типом `T&`. В строках 4–5 определяются простые требования, указывающие, что `t` будет считаться диапазоном, если мы можем получить доступ к начальному итератору и ограничителю, передав параметр `t` функциям `std::ranges::begin` и `std::ranges::end` соответственно. Если ни одно из этих простых требований не компилируется, параметр `t` не будет считаться диапазоном.

Простые требования могут применяться к выражениям с операторами. Следующий концепт определяет допустимые операции с целыми числами или с числами с плавающей точкой:

[1] Стандарт C++, «24.4.2. Ranges». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/ranges#range.range.

```
template<class T>
concept ArithmeticType =
   requires(T a, T b) {
      a + b;
      a - b;
      a * b;
      a / b;
      a += b;
      a -= b;
      a *= b;
      a /= b;
   };
```

Мы не указали оператор модуля (`%`), потому что он совместим только с целочисленными операндами. Конечно, стандартные арифметические типы поддерживают больше операций, например неявные преобразования между типами, поэтому «настоящий» концепт `ArithmeticType` содержал бы больше строк кода. Признак типа `is_arithmetic` проверяет стандартные арифметические типы. Для типов пользовательских классов он всегда возвращает `false`.

Требование к типу

Требование к типу (type requirement) начинается с ключевого слова `typename` и последующего имени типа и определяет, является ли указанный тип допустимым. Например, если ваш код требует, чтобы аргумент типа имел вложенный тип с именем `value_type` (в случае с контейнером стандартной библиотеки), вы можете определить концепт с требованием к типу:

```
template<typename T>
concept HasValueType = requires {
   typename T::value_type;
};
```

Тип `T` соответствует требованию `HasValueType`, если `T::value_type` является допустимым типом. Если тип `T` не содержит вложенного `value_type`, это требование будет оценено как `false`.

Составное требование

Составное требование (compound requirement) позволяет указать выражение, к результату которого также предъявляются требования. Составное требование имеет следующий вид:

```
{ выражение } -> требование-к-результату
```

Например, концепт `incrementable` из заголовка `<iterator>` стандартной библиотеки содержит выражение `requires`, указывающее, что итератор `incrementable` должен поддерживать оператор постфиксного инкремента (`++`)[1]:

[1] Стандарт C++, «23.3.4.5. Concept incrementable». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/iterator.concepts#iterator.concept.inc.

```
requires(I i) {
   { i++ } -> same_as<I>;
}
```

Обозначение `->` указывает на требование к результату выражения `i++`. В этом случае результат должен иметь тот же тип (`I`), что и тип объекта `i`. После правой фигурной скобки в составном требовании можно добавить ключевое слово `noexcept`, указывающее, что выражение не должно генерировать исключения. Когда мы использовали `std::same_as`, мы оперировали двумя аргументами шаблона, но здесь используем только один. Когда вы накладываете ограничение на результат выражения, компилятор вставляет тип этого выражения в ограничение в качестве первого параметра-типа. Поэтому в нашем ограничении на самом деле присутствуют два параметра-типа — в данном случае оба являются типом `I`[1].

Объект `incrementable` также должен поддерживать концепт `weakly_incrementable`, который, помимо других требований, содержит следующее составное требование:

```
{ ++i } -> same_as<I&>;
```

Таким образом, объект `incrementable` должен поддерживать оператор префиксного инкремента, а результат выражения должен иметь тот же тип, что и ссылка на параметр-тип `I`.

Вложенное требование

Вложенное требование (nested requirement) — это условие `requires`, вложенное в выражение `requires`. Вложенные требования используются для применения существующих концептов и признаков типов к параметрам выражения `requires`.

Разовое ограничение requires requires

Выражение `requires`, помещенное непосредственно в условие `requires`, — это **разовое ограничение (ad-hoc constraint)**, также называемое **специальным ограничением**. В следующем шаблоне функции `printRange` мы скопировали выражение `requires` концепта `std::ranges::range` и поместили его непосредственно в условие `requires`:

```
template<typename T>
   requires requires(const T& t) {
      std::ranges::begin(t);
      std::ranges::end(t);
   }
void printRange(const T& range) {
   for (const auto& item : range) {
      std::cout << item << " ";
   }
}
```

[1] Стандарт C++, «13.2. Template Parameters». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/temp.param#4.

Обозначение `requires requires` допустимо, если:

- первое требование является условием `requires`;
- второе требование является выражением `requires`.

Преимущество разового ограничения заключается в том, что его можно определить прямо в том месте кода, где оно используется, если оно нужно вам только один раз. Именованные концепты предпочтительнее.

15.6.6. Демонстрационные концепты

В документе стандарта C++ более 400 раз встречаются примечания «только в демонстрационных целях» («for the sake of exposition», «exposition only»). Они выделены курсивом и указывают на элементы языка, которые пока только обсуждаются[1]. Часто такие фрагменты стандарта демонстрируют, что и как может быть реализовано в будущем. Например, там упоминается демонстрационный концепт `has-arrow` библиотеки `ranges`, описывающий тип итераторов, поддерживающих оператор `->`[2]:

```
template<class I>
   concept has-arrow =
      input_iterator<I> && (is_pointer_v<I> ||
         requires(I i) { i.operator->(); });
```

Другой пример из стандарта — демонстрационный концепт `decrementable` библиотеки `ranges`, описывающий типы итераторов, поддерживающих оператор `--`[3]:

```
template<class I>
   concept decrementable =
      incrementable<I> && requires(I i) {
         { --i } -> same_as<I&>;
         { i-- } -> same_as<I>;
      };
```

У вас может появиться вопрос: почему концепт `decrementable` требует, чтобы тип соответствовал концепту `incrementable`? Напомним, что все итераторы поддерживают оператор `++`, поэтому итератор `decrementable` должен также быть и `incrementable`.

Мы перечислили демонстрационные концепты в следующей таблице. Вы можете найти каждый из них в разделе «Index of library concepts» стандарта C++[4].

---

[1] «Exposition-Only in the C++ Standard?». Ответ от 28 декабря 2015 г. Был доступен: 15 сентября 2023 г. https://stackoverflow.com/questions/34493104/exposition-only-in-the-c-standard.

[2] Стандарт C++, «24.5.1. Helper Concepts». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/range.utility.

[3] Стандарт C++, «24.6.4.2. Class Template iota_view». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/range.factories#range.iota.view-2.

[4] Стандарт C++, «Index of Library Concepts». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/conceptindex.

Демонстрационные концепты		
Библиотека концептов C++20		
boolean-testable	same-as-impl	
boolean-testable-impl	weakly-equality-comparable-with	
Библиотека итераторов		
can-reference	cpp17-input-iterator	dereferenceable
cpp17-bidirectional-iterator	cpp17-iterator	indirectly-readable-impl
cpp17-forward-iterator	cpp17-random-access-iterator	simple-view
Библиотека ranges C++20		
advanceable	has-tuple-element	pair-like-convertible-from
convertible-to-non-slicing	iterator-sentinel-pair	stream-extractable
decrementable	not-same-as	tiny-range
has-arrow	pair-like	
Сравнения в библиотеке языковой поддержки		
compares-as	partially-ordered-with	
Библиотека памяти		
no-throw-forward-iterator	no-throw-input-iterator	no-throw-sentinel
no-throw-forward-range	no-throw-input-range	

15.6.7. Технологии, применявшиеся до появления концептов: SFINAE и отправка тегов

С каждой новой версией стандарта C++ метапрограммирование становилось более мощным и удобным. К появлению концептов C++20 привел целый ряд технологий, включая SFINAE, отправку тегов и инструкции `constexpr if` (в разделе 15.13.3 мы рассмотрели `constexpr`). Хороший обзор истории их развития вы найдете в статье блога «Notes on C++ SFINAE, Modern C++ and C++20 Concepts»[1].

SFINAE — «неудавшаяся подстановка не является ошибкой»

Раньше мы упоминали, что при вызове функции компилятор осуществляет поиск всех соответствующих этому вызову функций, известных как набор перегрузок.

[1] Бартломей Филипек (Bartlomiej Filipek). «Notes on C++ SFINAE, Modern C++ and C++20 Concepts», 20 апреля 2020 г. Был доступен: 15 сентября 2023 г. https://www.bfilipek.com/2016/02/notes-on-c-sfinae.html.

Из этих перегрузок компилятор выбирает наилучшее соответствие. Этот процесс часто включает в себя создание специализации шаблона функции на основе типов аргументов вызова.

До появления концептов разработчики должны были значительно лучше знать правила специализации шаблонов. Поскольку отладка шаблона нетривиальна, это может приводить к появлению ошибок. Методы метапрограммирования на основе шаблонов, включающие `std::enable_if`, обычно использовались с признаками типов, чтобы проверять, соответствует ли аргумент шаблона функции требованиям шаблона. Если нет, компилятор создаст недопустимый код, который затем будет удален из набора перегрузок. Концепция SFINAE («substitution failure is not an error» — «неудавшаяся подстановка не является ошибкой»)[1,2,3] описывает, как компилятор отклоняет недопустимый код шаблона при поиске правильной функции для вызова. SFINAE не позволяет компилятору немедленно генерировать потенциально длинные списки ошибок при первом создании специализации шаблона. Компилятор генерирует сообщения об ошибках только в том случае, если он не может найти соответствие вызову функции в наборе перегрузок.

Отправка тегов

Вы можете указать компилятору версию перегруженной функции для вызова, основываясь не только на параметрах-типах шаблона, но и на свойствах этих типов, с помощью метода **отправки тегов (tag-dispatch)**[4]. Бьерн Страуструп в своей статье «Concepts: The Future of Generic Programming» называет свойства типов «концептами» и объясняет, как перегрузка с концептами C++20 (раздел 15.6.4) может заменить отправку тегов[5].

15.7. Тест-драйв концептов с помощью объявления static_assert

Концепты создают на этапе компиляции значения `bool`, которые можно тестировать на этапе компиляции с помощью объявления `static_assert`[6] (C++11). Оно было разработано, чтобы реализовать поддержку утверждений на этапе

---

[1] «Substitution failure is not an error». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Substitution_failure_is_not_an_error.

[2] Бартломей Филипек (Bartlomiej Filipek), «Notes on C++ SFINAE, Modern C++ and C++20 Concepts».

[3] Давид Вандевурд (Daveed Vandevoorde) и Николай М. Хосуттис (Nicolai M. Josuttis), «C++ Templates: The Complete Guide». Изд. Addison-Wesley Professional, 2002.

[4] «Generic Programming — Tag Dispatching». Был доступен: 15 сентября 2023 г. https://www.boost.org/community/generic_programming.html#tag_dispatching.

[5] Бьерн Страуструп (Bjarne Stroustrup), «Concepts: The Future of Generic Programming» (раздел 6). Был доступен: 15 сентября 2023 г. https://www.stroustrup.com/good_concepts.pdf.

[6] C++ Core Guidelines, «T.150: Check That a Class Matches a Concept Using static_assert». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-check-class.

компиляции и генерировать сообщения о неправильном использовании библиотеки шаблонов[1]. На илл. 15.9 показана программа, которая тестирует наш пользовательский концепт `Numeric` из раздела 15.6.1 в шаблоне функции `multiply`, не ограниченном концептами[2].

```
// fig15_09.cpp
// Тестирование пользовательского концепта с объявлением static_assert
#include <iostream>
#include <string>

template<typename T>
concept Numeric = std::integral<T> || std::floating_point<T>;

template<typename T>
auto multiply(T a, T b) {
   static_assert(Numeric<T>);
   return a * b;
}

int main() {
   using namespace std::string_literals;
   multiply(2, 5); // OK: int - это Numeric
   multiply(2.5, 5.5); // OK: double - это Numeric
   multiply("2"s, "5"s); // Ошибка: string - это не Numeric
}
```

```
fig15_09.cpp:11:4: error: static_assert failed
   static_assert(Numeric<T>);
   ^             ~~~~~~~~~~

fig15_09.cpp:19:4: note: in instantiation of function template specialization
'multiply<std::basic_string<char>>' requested here
   multiply("2"s, "5"s); // error: string is not Numeric
   ^

fig15_09.cpp:11:18: note: because 'std::basic_string<char>' does not satisfy
'Numeric'
   static_assert(Numeric<T>);
                 ^

fig15_09.cpp:7:24: note: because 'std::basic_string<char>' does not satisfy
'integral'
concept Numeric = std::integral<T> || std::floating_point<T>;
                       ^
```

[1] Роберт Кларер (Robert Klarer), Джон Мэддок (John Maddock), Беман Доуз (Beman Dawes) и Говард Хиннант (Howard Hinnant), «Proposal to Add Static Assertions to the Core Language (Revision 3)», 20 октября 2004 г. Был доступен: 15 сентября 2023 г. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1720.html.

[2] В практической работе с концептами вам не нужно использовать static_assert, как в этом примере.

```
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/concepts:
102:24: note: because 'is_integral_v<std::basic_string<char> >'
evaluated to false
    concept integral = is_integral_v<_Tp>;
                       ^

fig15_09.cpp:7:44: note: and 'std::basic_string<char>' does not satisfy
'floating_point'
concept Numeric = std::integral<T> || std::floating_point<T>;
                                           ^

/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/concepts:
111:30: note: because 'is_floating_point_v<std::basic_string<char> >'
evaluated to false
    concept floating_point = is_floating_point_v<_Tp>;
                             ^

fig15_09.cpp:12:13: error: invalid operands to binary expression
('std::basic_string<char>' and 'std::basic_string<char>')
   return a * b;
          ~ ^ ~

2 errors generated.
```

Илл. 15.9 | Тестирование пользовательского концепта с объявлением static_assert

Когда аргумент `static_assert` равен `false`, компилятор выводит сообщение об ошибке, которое поясняет, где находится ошибка и в чем она заключается. Если аргумент `static_assert` равен `true`, компилятор не выводит никаких сообщений и продолжает компилировать код. Выражение в строке 11

```
static_assert(Numeric<T>);
```

проверяет соответствие типа аргумента (`T`) требованиям концепта `Numeric`. Если требования удовлетворены, `Numeric<T>` вычисляется как `true` и компилятор продолжает компиляцию кода. Когда мы вызываем `multiple` в строках 17 и 18, `Numeric<T>` вычисляется как `true`, поскольку типы аргументов (`int` и `double`) соответствуют концепту `Numeric`.

Но когда мы вызываем функцию `multiply` в строке 19 со строковыми объектными литералами, `Numeric<T>` в строке 11 вычисляется как `false`, поскольку тип `string` не соответствует требованиям концепта. В выходных данных мы показываем сообщения об ошибках от компилятора Clang C++. Мы выделили главные фрагменты сообщений жирным шрифтом и добавили пустые строки для удобства чтения.

Для концепта `Numeric` компилятор сообщает:

```
note: because 'std::basic_string<char>' does not satisfy 'Numeric'
```

Сообщения также содержат более подробную информацию:

```
note: because 'std::basic_string<char>' does not satisfy 'integral'
```

а также:

```
note: and 'std::basic_string<char>' does not satisfy 'floating_point'
```

Объявление `static_assert` может содержать опциональный второй аргумент: строку, в которую будет записано сообщение об ошибке в случае ее появления.

15.8. Создание пользовательского алгоритма

В главах 13 и 14 вы узнали, что стандартная библиотека предоставляет контейнеры, итераторы и алгоритмы. Мы рассмотрели алгоритмы, работающие с элементами контейнеров с помощью итераторов. Вы можете воспользоваться преимуществами этой архитектуры, чтобы определить пользовательские алгоритмы, совместимые с любым контейнером, который поддерживает их требования к итераторам. На илл. 15.10 определяется ограниченный алгоритм `average`, аргумент которого должен соответствовать пользовательскому концепту `NumericInputRange`, определенному в строках 14–17.

```
 1 // fig15_10.cpp
 2 // Пользовательский алгоритм, вычисляющий среднее арифметическое
 3 // нескольких чисел (элементов диапазона)
 4 #include <algorithm>
 5 #include <array>
 6 #include <concepts>
 7 #include <iostream>
 8 #include <iterator>
 9 #include <list>
10 #include <ranges>
11 #include <vector>
12
13 // Концепт для диапазона, содержащего целые числа или числа с плавающей точкой
14 template<typename T>
15 concept NumericInputRange = std::ranges::input_range<T> &&
16    (std::integral<typename T::value_type> ||
17    std::floating_point<typename T::value_type>);
18
19 // Вычисляем среднее арифметическое элементов диапазона NumericInputRange
20 auto average(NumericInputRange auto const& range) {
21    long double total{0};
22
23    for (auto i{range.begin()}; i != range.end(); ++i) {
24       total += *i; // Разыменовываем итератор и прибавляем число к total
25    }
26
27    // Делим total на количество элементов в диапазоне
28    return total / std::ranges::distance(range);
29 }
30
31 int main() {
```

```
32    std::ostream_iterator<int> outputInt(std::cout, " ");
33    const std::array ints{1, 2, 3, 4, 5};
34    std::cout << "array ints: ";
35    std::ranges::copy(ints, outputInt);
36    std::cout << "\naverage of ints: " << average(ints);
37
38    std::ostream_iterator<double> outputDouble(std::cout, " ");
39    const std::vector doubles{10.1, 20.2, 35.3};
40    std::cout << "\n\nvector doubles: ";
41    std::ranges::copy(doubles, outputDouble);
42    std::cout << "\naverage of doubles: " << average(doubles);
43
44    std::ostream_iterator<long double> outputLongDouble(std::cout, " ");
45    const std::list longDoubles{10.1L, 20.2L, 35.3L};
46    std::cout << "\n\nlist longDoubles: ";
47    std::ranges::copy(longDoubles, outputLongDouble);
48    std::cout << "\naverage of longDoubles: " << average(longDoubles)
49       << "\n";
50 }
```

```
array ints: 1 2 3 4 5
average of ints: 3

vector doubles: 10.1 20.2 35.3
average of doubles: 21.8667

list longDoubles: 10.1 20.2 35.3
average of doubles: 21.8667
```

Илл. 15.10 | Пользовательский алгоритм, вычисляющий среднее арифметическое нескольких чисел (элементов диапазона)

Пользовательский концепт NumericInputRange

Пользовательский концепт `NumericInputRange` (строки 14–17) проверяет:

- соответствие типа концепту `input_range` (строка 15); то есть аргумент должен поддерживать как минимум итераторы ввода для чтения элементов;
- соответствие элементов диапазона концепту `std::integral` или `std::floating_point` (строки 16–17) — только такие типы можно использовать в вычислениях.

Напомним из раздела 13.2.1, что каждый контейнер стандартной библиотеки имеет вложенный `value_type`, указывающий тип элементов контейнера. Мы используем его для проверки соответствия элементов требованиям концепта. В строках 16 и 17 обозначение

```
typename T::value_type
```

указывает, что `T::value_type` является псевдонимом для типа элементов диапазона.

Пользовательский алгоритм average

В строках 20–29 определяется алгоритм `average` как сокращенный шаблон функции. Мы ограничили его параметр `range` пользовательским концептом `NumericInputRange`, поэтому алгоритм может работать с любым `input_range`, содержащим числа. Чтобы гарантировать поддержку алгоритмом `average` любого стандартного числового типа, мы используем переменную типа `long double` (строка 21) для хранения суммы элементов.

В строках 23–25 мы проходим по каждому элементу диапазона от начального итератора до конечного итератора (не включая конечный итератор), прибавляя значение каждого элемента к `total`. Затем в строке 28 мы делим количество элементов диапазона на количество элементов, найденное алгоритмом `std::ranges::distance`.

Пользовательский алгоритм average и контейнеры стандартной библиотеки

Чтобы показать возможность обработки разных контейнеров стандартной библиотеки нашим алгоритмом `average`, в строках 33, 39 и 45 мы определяем три контейнера: `array` с элементами `int`, `vector` с элементами `double` и `list` с элементами `long double`. В строках 36, 42 и 48 мы вызываем алгоритм `average` для каждого из этих контейнеров, вычисляя среднее арифметическое его элементов.

15.9. Создание пользовательского контейнера и пользовательских итераторов

В главе 11 мы создали класс `myArray` для демонстрации специальных функций и перегрузки операторов. В C++ Core Guidelines рекомендуется использовать шаблоны для реализации любого класса, представляющего контейнер значений или диапазон значений[1]. Поэтому в этом разделе мы определим `myArray` как шаблон класса и улучшим его, используя соглашения стандартной библиотеки[2]. Мы также определим пользовательские итераторы, позволяющие использовать объекты `myArray` с различными алгоритмами стандартной библиотеки.

Наша цель — дать вам представление о том, как создавать контейнеры, подобные контейнерам стандартной библиотеки. Для достижения полной совместимости со стандартной библиотекой и обратной совместимости с предыдущими версиями языка C++ требуется множество директив условной компиляции, выходящих за рамки этой книги. Если вы хотите создать многократно используемые контейнеры, подобные контейнерам стандартной библиотеки, изучите код, предоставляемый

---

[1] C++ Core Guidelines, «T.3: Use Templates to Express Containers and Ranges». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-cont.

[2] Джонатан Боккара (Jonathan Boccara), «Make Your Containers Follow the Conventions of the STL», 24 апреля 2018 г. Был доступен: 15 сентября 2023 г. https://www.fluentcpp.com/2018/04/24/following-conventions-stl/.

поставщиками компиляторов, и ознакомьтесь с источниками, приведенными в наших сносках. На илл. 15.11 определяется наш шаблон класса `myArray` и его пользовательские итераторы. Мы разделили код на несколько фрагментов для удобства обсуждения.

Один файл заголовка

Одно из заметных изменений по сравнению с классом `myArray` из главы 11 заключается в том, что весь класс и его пользовательские итераторы определены в одном заголовке. Это вполне типично для шаблонов классов. Компилятору необходимо полное определение для создания специализации шаблона. Кроме того, определение функций внутри определения шаблона класса упрощает синтаксис, поскольку вам не нужны заголовки `template` для каждой функции.

myArray поддерживает двунаправленные итераторы

Мы смоделировали этот пример по образцу класса `array` стандартной библиотеки, который использует традиционный массив фиксированного размера, выделяемый на этапе компиляции. Как вы помните из раздела 13.3, указатели таких массивов соответствуют всем требованиям итераторов произвольного доступа. Однако для этого примера мы реализуем пользовательские двунаправленные итераторы, используя шаблоны классов. Архитектура итераторов, которую мы используем, была вдохновлена реализацией класса `array`[1] в стандартной библиотеке Microsoft C++ с открытым исходным кодом, определяющей два класса итераторов:

- класс итераторов, работающих с константными объектами;
- класс итераторов, работающих с изменяемыми объектами.

Здесь вы можете ознакомиться с реализацией класса `array` от Microsoft:

https://github.com/microsoft/STL/blob/main/stl/inc/array

Мы определим следующие пользовательские классы итераторов:

- класс `ConstIterator` представляет двунаправленный итератор, предназначенный только для чтения;
- класс `Iterator` представляет двунаправленный итератор, предназначенный для чтения и записи.

Класс `array` в реализациях GNU и Clang просто использует указатели для итераторов массива. Здесь вы можете ознакомиться с этими реализациями:

https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/std/array

https://github.com/llvm/llvm-project/blob/main/libcxx/include/array

---

[1] «array Standard Header», 24 февраля 2021 г. Был доступен: 15 сентября 2023 г. https://github.com/microsoft/STL/blob/main/stl/inc/array.

Почему мы используем двунаправленные итераторы, а не итераторы произвольного доступа

Напомним из нашего введения в тему итераторов в разделе 13.3, что контейнеры стандартной библиотеки поддерживают различные уровни итераторов. Наиболее функциональные — непрерывные итераторы, которые представляют собой итераторы произвольного доступа для контейнеров, гарантирующих размещение элементов в непрерывной области памяти. Большинство алгоритмов стандартной библиотеки работает с диапазонами или с элементами контейнеров. Только 12 алгоритмов требуют итераторов произвольного доступа: алгоритм `shuffle` и 11 алгоритмов сортировки, и ни один из этих 12 алгоритмов не требует непрерывных итераторов. Двунаправленные итераторы класса `myArray` позволяют большинству алгоритмов стандартной библиотеки обрабатывать объекты этого класса. К итераторам произвольного доступа предъявляется намного больше требований: они должны поддерживать целый ряд операций[1], показанных в следующей таблице. В качестве упражнения вы можете модернизировать итераторы класса `myArray`, добавив поддержку этих операций, чтобы получить итераторы произвольного доступа.

Операция с итератором произвольного доступа	Описание
`p += i`	Перемещает итератор `p` вперед (выполняет инкремент) на `i` позиций
`p -= i`	Перемещает итератор `p` назад (выполняет декремент) на `i` позиций
`p + i` *или* `i + p`	Результат — итератор, расположенный на позиции `p + i`
`p - i`	Значение выражения — итератор, расположенный на позиции `p - i`
`p - p1`	Результат — целое число, обозначающее расстояние (то есть количество элементов) между двумя элементами в одном контейнере
`p[i]`	Возвращает ссылку на элемент, смещенный от `p` на `i` позиций вперед
`p < p1`	Возвращает `true`, если итератор `p` меньше итератора `p1` (то есть если `p` находится ближе к началу контейнера, чем `p1`); в противном случае возвращает `false`
`p <= p1`	Возвращает `true`, если итератор `p` меньше или равен итератору `p1` (то есть если `p` находится не дальше от начала контейнера, чем `p1`); в противном случае возвращает `false`

[1] Стандарт C++, «23.3.4.13. Concept random_access_iterator». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/iterator.concept.random.access.

Операция с итератором произвольного доступа	Описание
`p > p1`	Возвращает `true`, если итератор `p` больше итератора `p1` (то есть если `p` находится дальше от начала контейнера, чем `p1`); в противном случае возвращает `false`
`p >= p1`	Возвращает `true`, если итератор `p` больше или равен итератору `p1` (то есть если `p` находится не ближе к началу контейнера, чем `p1`); в противном случае возвращает `false`

Основные требования к итераторам

Все итераторы должны поддерживать[1, 2, 3]:

- вызов конструктора по умолчанию;
- вызов копирующего конструктора;
- копирующее присваивание;
- вызов деструктора;
- обмен значениями.

Мы реализуем классы итераторов, используя «правило нуля» (раздел 11.6.6) и позволяя компилятору генерировать копирующий конструктор, оператор копирующего присваивания, перемещающий конструктор, оператор перемещающего присваивания и специальные функции деструктора.

15.9.1. Шаблон класса ConstIterator

В строках 15–77 на илл. 15.11 определен шаблон класса `ConstIterator`, который используется классом `myArray` для создания итераторов, предназначенных только для чтения. Шаблон имеет один параметр-тип `T` (строка 15), представляющий тип элемента `myArray`. Каждый `ConstIterator` указывает на элемент этого типа.

```
1 // Fig. 15.11: MyArray.h
2 // Шаблон класса MyArray с пользовательскими итераторами,
3 // реализованными по шаблонам классов ConstIterator и Iterator
4 #pragma once
5 #include <algorithm>
6 #include <compare>
```

Продолжение ⇨

[1] «C++ Named Requirements: LegacyIterator». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/named_req/Iterator.

[2] Triangles, «Writing a Custom Iterator in Modern C++», 19 декабря 2020 г. Был доступен: 15 сентября 2023 г. https://internalpointers.com/post/writing-custom-iterators-modern-cpp.

[3] Дэвид Горски (David Gorski), «Custom STL Compatible Iterators», 2 марта 2019 г. Был доступен: 15 сентября 2023 г. https://davidgorski.ca/posts/stl-iterators/.

```
#include <initializer_list>
#include <iostream>
#include <iterator>
#include <memory>
#include <stdexcept>
#include <utility>

// Шаблон класса ConstIterator для константного итератора класса MyArray
template <typename T>
class ConstIterator {
public:
   // Вложенные типы открытых итераторов
   using iterator_category = std::bidirectional_iterator_tag;
   using difference_type = std::ptrdiff_t;
   using value_type = T;
   using pointer = const value_type*;
   using reference = const value_type&;

   // Конструктор по умолчанию
   ConstIterator() = default;

   // Инициализируем ConstIterator с помощью указателя на MyArray
   ConstIterator(pointer p) : m_ptr{p} {}

   // ОПЕРАЦИИ, КОТОРЫЕ ДОЛЖНЫ ПРЕДОСТАВЛЯТЬ ВСЕ ИТЕРАТОРЫ
   // Выполняем инкремент итератора к следующему элементу
   // и возвращаем ссылку на итератор
   ConstIterator& operator++() noexcept {
      ++m_ptr;
      return *this;
   }

   // Выполняем инкремент итератора к следующему элементу
   // и возвращаем итератор до этого инкремента
      ConstIterator operator++(int) noexcept {
      ConstIterator temp{*this};
      ++(*this);
      return temp;
   }

   // ОПЕРАЦИИ, КОТОРЫЕ ДОЛЖНЫ ПРЕДОСТАВЛЯТЬ ВСЕ ИТЕРАТОРЫ ВВОДА
   // Возвращаем константную ссылку на элемент, на который указывает m_ptr
   reference operator*() const noexcept {return *m_ptr;}

   // Возвращаем константный указатель на элемент, на который указывает m_ptr
   pointer operator->() const noexcept {return m_ptr;}

   // Оператор <=> автоматически поддерживает операторы равенства и сравнения
   // Двунаправленным итераторам требуются только операторы == и !=
   // Эта реализация поддерживает необходимые итераторам произвольного
   // доступа операторы <, <=, > и >=
```

```
58    auto operator<=>(const ConstIterator& other) const = default;
59
60    // ОПЕРАЦИИ, КОТОРЫЕ ДОЛЖНЫ ПРЕДОСТАВЛЯТЬ ВСЕ ДВУНАПРАВЛЕННЫЕ ИТЕРАТОРЫ
61    // Выполняем декремент итератора к предыдущему элементу
62    // и возвращаем ссылку на итератор
63    ConstIterator& operator--() noexcept {
64       --m_ptr;
65       return *this;
66    }
67
68    // Выполняем декремент итератора к предыдущему элементу
69    // и возвращаем итератор до этого декремента
70    ConstIterator operator--(int) noexcept {
71       ConstIterator temp{*this};
72       --(*this);
73       return temp;
74    }
75 private:
76    pointer m_ptr{nullptr};
77 };
78
```

Илл. 15.11 (часть 1 из 4) | Шаблон класса MyArray с пользовательскими итераторами, реализованными по шаблонам классов ConstIterator и Iterator — класс ConstIterator

Стандартные имена вложенных типов

В строках 19–23 определяются псевдонимы имен вложенных типов, предусмотренные в классах итераторов стандартной библиотеки C++[1]:

- `iterator_category` — категория итераторов (раздел 13.3.2), указанная здесь как тип `std::bidirectional_iterator_tag` из заголовка `<iterator>`. Этот «теговый» тип указывает на двунаправленные итераторы. Алгоритмы стандартной библиотеки используют признаки типов и концепты C++20, чтобы подтверждать корректность категории итераторов контейнера для использования с каждым алгоритмом.
- `difference_type` — тип результата вычитания одного `ConstIterator` из другого; `std::ptrdiff_t` — это тип результата вычитания указателей.
- `value_type` — тип элемента, на который указывает `ConstIterator`.
- `pointer` — тип указателя на константный объект `value_type`. Элемент данных `ConstIterator` (строка 76) объявляется с этим типом.
- `reference` — тип ссылки на константный объект `value_type`.

[1] Стандарт C++, «23.3.2.3. Iterator Traits». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/iterators#iterator.traits.

Конструкторы

Класс `ConstIterator` предоставляет конструктор по умолчанию без аргументов (строка 26), который инициализирует элемент `m_ptr`, используя внутриклассовый инициализатор (`nullptr`; строка 76), и конструктор, который инициализирует `ConstIterator` с помощью указателя на элемент (строка 29). Копирующий и перемещающий конструкторы класса генерируются автоматически.

Операторы ++

В строках 34–37 и 41–45 определяются операторы префиксного и постфиксного инкремента, которые должны предоставлять все итераторы. Оператор префиксного инкремента устанавливает `m_ptr` на следующий элемент и возвращает ссылку на итератор. Оператор постфиксного инкремента устанавливает `m_ptr` на следующий элемент и возвращает копию итератора, созданную перед выполнением инкремента.

Перегруженные операторы * и ->

Семантически итераторы похожи на указатели, поэтому они должны перегружать операторы `*` и `->` (строки 49 и 52). Перегруженный оператор `*` разыменовывает `m_ptr` для доступа к элементу, на который в данный момент указывает итератор, и возвращает ссылку на этот элемент. Перегруженный оператор `->` возвращает `m_ptr` в качестве указателя на этот элемент.

Сравнение двунаправленных итераторов

Двунаправленные итераторы должны быть сравнимы с помощью операторов равенства `==` и `!=`. Здесь мы используем сгенерированный компилятором оператор трехстороннего сравнения `<=>` (строка 58) для поддержки операций сравнения итераторов `ConstIterator`. Если вы доработаете наши классы итераторов, чтобы сделать их итераторами произвольного доступа, эта реализация позволит выполнять операции сравнения операторами `<`, `<=`, `>` и `>=` в соответствии с требованиями итераторов произвольного доступа.

Операторы --

В строках 63–66 и 70–74 определяются операторы префиксного и постфиксного декремента, которые должны предоставлять все двунаправленные итераторы. Эти операторы работают аналогично операторам инкремента, но перемещают `m_ptr` на предыдущий элемент.

15.9.2. Шаблон класса Iterator

В строках 81–137 на илл. 15.11 определяется шаблон класса `Iterator`, который класс `myArray` использует для создания итераторов, предназначенных для чтения и записи. Шаблон имеет один параметр-тип `T` (строка 81), представляющий тип элемента `myArray`. Класс `Iterator` является производным от класса `ConstIterator<T>` (строка 82).

```
79 // Шаблон класса Iterator для неконстантного итератора класса MyArray;
80 // переопределяет несколько унаследованных операторов для возврата изменяемых
   результатов
81 template <typename T>
82 class Iterator : public ConstIterator<T> {
83 public:
84    // Вложенные типы открытых итераторов
85    using iterator_category = std::bidirectional_iterator_tag;
86    using difference_type = std::ptrdiff_t;
87    using value_type = T;
88    using pointer = value_type*;
89    using reference = value_type&;
90
91    // Наследуем конструкторы ConstIterator
92    using ConstIterator<T>::ConstIterator;
93
94    // ОПЕРАЦИИ, КОТОРЫЕ ДОЛЖНЫ ПРЕДОСТАВЛЯТЬ ВСЕ ИТЕРАТОРЫ
95    // Выполняем инкремент итератора к следующему элементу
96    // и возвращаем ссылку на итератор
97    Iterator& operator++() noexcept {
98       ConstIterator<T>::operator++(); // Вызываем версию базового класса
99       return *this;
100    }
101
102    // Выполняем инкремент итератора к следующему элементу
103    // и возвращаем итератор до этого инкремента
104    Iterator operator++(int) noexcept {
105       Iterator temp{*this};
106       ConstIterator<T>::operator++(); // Вызываем версию базового класса
107       return temp;
108    }
109
110    // ОПЕРАЦИИ, КОТОРЫЕ ДОЛЖНЫ ПРЕДОСТАВЛЯТЬ ВСЕ ИТЕРАТОРЫ ВВОДА
111    // Возвращаем ссылку на элемент, на который указывает m_ptr; этот оператор
112    // возвращает неконстантную ссылку для поддержки итератора вывода
113    reference operator*() const noexcept {
114       return const_cast<reference>(ConstIterator<T>::operator*());
115    }
116
117    // Возвращаем указатель на элемент, на который указывает m_ptr
118    pointer operator->() const noexcept {
119       return const_cast<pointer>(ConstIterator<T>::operator->());
120    }
121
122    // ОПЕРАЦИИ, КОТОРЫЕ ДОЛЖНЫ ПРЕДОСТАВЛЯТЬ ВСЕ ДВУНАПРАВЛЕННЫЕ ИТЕРАТОРЫ
123    // Выполняем декремент итератора к предыдущему элементу
124    // и возвращаем ссылку на итератор
125    Iterator& operator--() noexcept {
126       ConstIterator<T>::operator--(); // Вызываем версию базового класса
127       return *this;
128    }
129
```

Окончание ⇨

```
130     // Выполняем декремент итератора к предыдущему элементу
131     // и возвращаем итератор до этого декремента
132     Iterator operator--(int) noexcept {
133        Iterator temp{*this};
134        ConstIterator<T>::operator--(); // Вызываем версию базового класса
135        return temp;
136     }
137 };
138
```

Илл. 15.11 (часть 2 из 4) | Шаблон класса MyArray с пользовательскими итераторами, реализованными по шаблонам классов ConstIterator и Iterator — класс Iterator

Стандартные имена вложенных типов

В строках 85–89 определяются псевдонимы имен вложенных типов, предусмотренные в классах итераторов стандартной библиотеки C++. Шаблон класса `Iterator` определяет итераторы для чтения и записи, поэтому в строках 88–89 определяются псевдонимы `pointer` и `reference` без ключевого слова `const`. Указатели и ссылки данных типов могут использоваться для записи новых значений в элементы.

Конструкторы

Конструкторы класса `ConstIterator` знают, как инициализировать унаследованный элемент `m_ptr` класса `Iterator`, поэтому в строке 92 мы просто наследуем конструкторы базового класса.

Операторы ++ и --

В строках 97–100, 104–108, 125–128 и 132–136 определяются операторы префиксного и постфиксного инкремента и декремента. Каждый из них вызывает соответствующую версию `ConstIterator`. Префиксные операторы возвращают ссылку на обновленный итератор, а постфиксные операторы возвращают копию итератора, созданную перед выполнением операции.

Перегруженные операторы * и ->

В строках 113–115 и 118–120 определены перегруженные операторы * и ->. Каждый из них вызывает версию `ConstIterator`. Эти версии возвращают указатель или ссылку, которые рассматривают `value_type` как константу. Поскольку итераторы должны предоставлять возможность чтения и записи, в строках 114 и 119 используется оператор `const_cast` для возможности изменения указателя или ссылки (то есть для возможности удаления ключевого слова `const`), возвращенных перегруженными операторами базового класса.

15.9.3. Шаблон класса MyArray

В строках 141–211 на илл. 15.11 определяется упрощенный шаблон класса `myArray`. Мы удалили некоторые перегруженные операторы и специальные функции,

представленные в главе 11, чтобы сосредоточиться на контейнере и его итераторах. Чтобы имитировать шаблон класса `std::array`, мы определяем `myArray` как агрегатный тип (раздел 9.21), требующий, чтобы все нестатические данные были открытыми. Итак, мы определили `myArray`, используя структуру, которая по умолчанию имеет открытые данные. Кроме того, данные нового шаблона класса `myArray` хранятся в традиционном массиве фиксированного размера (строка 210), который создается на этапе компиляции вместо выделения динамической памяти во время выполнения.

```
139 // Шаблон класса MyArray, содержащий массив фиксированного размера T[SIZE];
140 // MyArray - агрегатный тип с открытыми данными наподобие std::array
141 template <typename T, size_t SIZE>
142 struct MyArray {
143    // Имена типов, используемые в контейнерах стандартной библиотеки
144    using value_type = T;
145    using size_type = size_t;
146    using difference_type = ptrdiff_t;
147    using pointer = value_type*;
148    using const_pointer = const value_type*;
149    using reference = value_type&;
150    using const_reference = const value_type&;
151
152    // Имена типов итераторов, используемые в контейнерах стандартной
       библиотеки
153    using iterator = Iterator<T>;
154    using const_iterator = ConstIterator<T>;
155    using reverse_iterator = std::reverse_iterator<iterator>;
156    using const_reverse_iterator = std::reverse_iterator<const_iterator>;
157
158    // Правило нуля: специальные функции MyArray генерируются автоматически
159
160    constexpr size_type size() const noexcept {return SIZE;} // Возвращаем
                                                                 размер
161
162    // Функции, возвращающие итераторы
163    iterator begin() {return iterator{&m_data[0]};}
164    iterator end() {return iterator{&m_data[0] + size()};}
165    const_iterator begin() const {return const_iterator{&m_data[0]};}
166    const_iterator end() const {
167       return const_iterator{&m_data[0] + size()};
168    }
169    const_iterator cbegin() const {return begin();}
170    const_iterator cend() const {return end();}
171
172    // Функции, возвращающие обратные итераторы
173    reverse_iterator rbegin() {return reverse_iterator{end()};}
174    reverse_iterator rend() {return reverse_iterator{begin()};}
175    const_reverse_iterator rbegin() const {
176       return const_reverse_iterator{end()};
177    }
178    const_reverse_iterator rend() const {
```

Окончание ⇨

```
179        return const_reverse_iterator(begin());
180     }
181     const_reverse_iterator crbegin() const {return rbegin();}
182     const_reverse_iterator crend() const {return rend();}
183
184     // Автоматически сгенерированный оператор трехстороннего сравнения
185     auto operator<=>(const MyArray& t) const noexcept = default;
186
187     // Перегруженный оператор индекса для неконстантных объектов MyArray;
188     // возвращаем ссылку на изменяемое значение lvalue
189     T& operator[](size_type index) {
190        // Проверяем корректность индекса
191        if (index >= size()) {
192           throw std::out_of_range("Index out of range");
193        }
194
195        return m_data[index]; // Возвращаем ссылку
196     }
197
198     // Перегруженный оператор индекса для константных объектов MyArray;
199     // возвращаем ссылку на неизменяемое значение lvalue
200     const T& operator[](size_type index) const {
201        // Проверяем корректность индекса
202        if (index >= size()) {
203           throw std::out_of_range("Index out of range");
204        }
205
206        return m_data[index]; // Возвращаем копию элемента
207     }
208
209     // Подобно std::array, данные являются открытыми, это агрегатный тип
210     T m_data[SIZE]; // Традиционный массив с типом T и размером SIZE
211 };
212
```

Илл. 15.11 (часть 3 из 4) | Шаблон класса MyArray с пользовательскими итераторами, реализованными по шаблонам классов ConstIterator и Iterator — класс MyArray

Заголовок template класса MyArray

Заголовок `template` (строка 141) показывает, что `myArray` является шаблоном класса. Этот заголовок задает два параметра:

- `T` представляет тип элемента `myArray`.
- `SIZE` — это параметр шаблона, не определяющий тип данных. Этот параметр обрабатывается как константа на этапе компиляции. Мы используем `SIZE` для представления количества элементов `myArray`.

В строке 210 создается традиционный массив элементов типа `T`. Количество этих элементов равно `SIZE`. Хотя мы здесь это не показали, параметры шаблона, не определяющие типы данных, могут иметь аргументы по умолчанию.

Стандартные имена вложенных типов

В строках 144–156 определяются псевдонимы имен вложенных типов, предусмотренные в классах контейнеров стандартной библиотеки C++. Типы обратных (реверсивных) итераторов специфичны для контейнеров, имеющих как минимум двунаправленные итераторы[1]:

- `value_type` — тип элементов контейнера (`T`);
- `size_type` — тип, представляющий количество элементов контейнера;
- `difference_type` — тип результата при вычитании итераторов;
- `pointer` — тип указателя на объект `value_type`;
- `const_pointer` — тип указателя на объект `const value_type`;
- `reference` — тип ссылки на объект `value_type`;
- `const_reference` — тип ссылки на объект `const value_type`;
- `iterator` — тип итератора для чтения и записи элементов `myArray` (`Iterator<T>`);
- `const_iterator` — тип итератора для чтения элементов `myArray` (`ConstIterator<T>`);
- `reverse_iterator` — тип обратного итератора для чтения и записи элементов `myArray`. Адаптер итератора `std::reverse_iterator` создает тип итератора из своего аргумента, который должен быть как минимум двунаправленным итератором;
- `const_reverse_iterator` — тип обратного итератора для чтения элементов `myArray`. Адаптер итератора `std::reverse_iterator` создает тип константного обратного итератора из своего аргумента, который должен быть как минимум двунаправленным итератором.

Функции класса MyArray, возвращающие итераторы

В строках 163–182 определяются функции класса `myArray`, возвращающие различные типы итераторов `myArray`, в том числе обратных. Важнейшие функции — `begin` и `end` из строк 163–168:

- `begin` (строка 163) возвращает итератор, указывающий на первый элемент `myArray`;
- `end` (строка 164) возвращает итератор, указывающий на следующий элемент после последнего элемента `myArray`;
- `begin` (строка 165) — перегруженная версия; возвращает константный итератор, указывающий на первый элемент `myArray`;

[1] Стандарт C++, «Table 73: Container Requirements». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/container.requirements#tab:container.req.

- `end` (строки 166–168) — перегруженная версия; возвращает константный итератор, указывающий на следующий элемент после последнего элемента `myArray`.

Другие функции, возвращающие итераторы, вызывают названные выше функции `begin` и `end`:

- `cbegin` и `cend` (строки 169–170) вызывают версии функций `begin` и `end`, возвращающие константные итераторы;
- `rbegin` и `rend` (строки 173–174) создают обратные итераторы на основе итераторов, возвращаемых неконстантными версиями функций `begin` и `end`;
- `rbegin` и `rend` (строки 175–180) — перегруженные версии; создают константные обратные итераторы на основе константных итераторов, возвращаемых константными версиями `begin` и `end`;
- `crbegin` и `crend` (строки 181–182) возвращают константные обратные итераторы, вызывая константные версии `rbegin` и `rend`.

Перегруженные операторы класса MyArray

В строках 185, 189–196 и 200–207 определяются перегруженные операторы класса `myArray`. Сгенерированный компилятором оператор трехстороннего сравнения (строка 185), который мы не будем использовать в этом примере, позволяет сравнивать целые объекты `myArray`, имеющие одинаковые типы элементов и размер. Перегруженные функции `operator[]` идентичны функциям класса `myArray` из главы 11, но теперь мы вызываем функцию `size` (строки 191 и 202), чтобы убедиться, что индекс не вышел за границы контейнера.

15.9.4. Дедуктивная инструкция для скобочной инициализации объекта myArray

Вы уже знаете, что можете инициализировать `std::array` с помощью CTAD. Предположим, что компилятор видит такую инструкцию:

```
std::array ints{1, 2, 3, 4, 5};
```

В этом случае компилятор автоматически определяет тип элементов массива как `int`, потому что все инициализаторы имеют тип `int`, и подсчитывает количество инициализаторов, чтобы определить размер массива. Класс `myArray` не предоставляет конструктор, принимающий произвольное количество аргументов. Поэтому мы написали **дедуктивную инструкцию (deduction guide)**[1] (строки 214–215), указывающую компилятору, как он должен определять тип `myArray` по скобочному инициализатору. Затем компилятор может использовать агрегатную инициализа-

[1] «Class Template Argument Deduction». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/language/class_template_argument_deduction.

цию (как в строке 15 на илл. 15.12), чтобы поместить инициализаторы в элементы данных традиционного массива, базового для `myArray`.

```
213 // Дедуктивная инструкция для скобочной инициализации объектов MyArray
214 template<typename T, std::same_as<T>... Us>
215 MyArray(T first, Us... rest) -> MyArray<T, 1 + sizeof...(Us)>;
```

Илл. 15.11 (часть 4 из 4) | Шаблон класса MyArray с пользовательскими итераторами, реализованными по шаблонам классов ConstIterator и Iterator — дедуктивная инструкция для MyArray

Дедуктивная инструкция — это шаблон. В его заголовке `template` используется синтаксис вариативного шаблона (`...`), который мы подробно рассмотрим в разделе 15.12. Строка 214 означает, что компилятор должен найти список инициализаторов, который:

- содержит хотя бы один инициализатор, представляющий `typename T`;
- может содержать любое количество дополнительных инициализаторов типа `T`, представляющих `std::same_as<T>... Us>`. Синтаксис `...` указывает на **пакет параметров (parameter pack)**. Имя пакета параметров (`Us`) обычно имеет множественное число (с окончанием `s`), поскольку оно может представлять несколько элементов.

Слева от оператора `->` в строке 215 мы указываем то, что похоже на начало конструктора `myArray`, получающего свой первый аргумент в параметре `T` с именем `first` и все остальные аргументы в параметре `Us...` с именем `rest`. Справа от оператора `->` мы сообщаем компилятору, что при инициализации `myArray` с помощью CTAD он должен «понять», что мы хотим создать `myArray` с элементами типа `T` и размером, указанным как:

```
1 + sizeof...(Us)
```

В этом выражении:

- `1` — минимальное количество инициализаторов в списке инициализаторов;
- `sizeof...(Us)` использует на этапе компиляции оператор `sizeof...` (C++11) для определения количества дополнительных инициализаторов, которые компилятор поместил в пакет параметров `Us`.

Наша дедуктивная инструкция основана на тех аналогах, которые предоставлены компиляторами GNU и Clang для их реализаций класса `std::array`. Здесь вы можете ознакомиться с дедуктивными инструкциями GNU и Clang в их соответствующих заголовках `<array>`:

https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/std/array

https://github.com/llvm/llvm-project/blob/main/libcxx/include/array

15.9.5. Тест-драйв класса myArray и его пользовательских итераторов с алгоритмами std::ranges

Программа, показанная на илл. 15.12, создает три объекта класса `MyArray`, содержащие значения `int`, `double` и `string` соответственно, а затем обрабатывает эти объекты несколькими алгоритмами `std::ranges`, которым нужны разные типы итераторов: ввода, вывода, однонаправленные и двунаправленные. Мы также используем цикл `for` для диапазона значений, чтобы поочередно обрабатывать элементы `myArray`. Класс `myArray` предоставляет двунаправленные итераторы, поэтому он совместим и с пользовательским алгоритмом `average` (илл. 15.10), которому нужны только итераторы вывода.

```
// fig15_12.cpp
// Использование MyArray с циклом for для диапазона значений
// и с алгоритмами стандартной библиотеки C++
#include <iostream>
#include <iterator>
#include "MyArray.h"

int main() {
   std::ostream_iterator<int> outputInt{std::cout, " "};
   std::ostream_iterator<double> outputDouble{std::cout, " "};
   std::ostream_iterator<std::string> outputString{std::cout, " "};

   std::cout << "Displaying MyArrays with std::ranges::copy, "
      << "which requires input iterators:\n";
   MyArray ints{1, 2, 3, 4, 5, 6, 7, 8};
   std::cout << "ints: ";
   std::ranges::copy(ints, outputInt);

   MyArray doubles{1.1, 2.2, 3.3, 4.4, 5.5};
   std::cout << "\ndoubles: ";
   std::ranges::copy(doubles, outputDouble);

   using namespace std::string_literals; // Строковые объектные литералы
   MyArray strings{"red"s, "orange"s, "yellow"s};
   std::cout << "\nstrings: ";
   std::ranges::copy(strings, outputString);

   std::cout << "\n\nDisplaying a MyArray with a range-based for "
      << "statement, which requires input iterators:\n";
   for (const auto& item : doubles) {
      std::cout << item << " ";
   }

   std::cout << "\n\nCopying a MyArray with std::ranges::copy, "
      << "which requires an input range and an output iterator:\n";
   MyArray<std::string, strings.size()> strings2{};
   std::ranges::copy(strings, strings2.begin());
   std::cout << "strings2 after copying from strings: ";
```

```
39    std::ranges::copy(strings2, outputString);
40
41    std::cout << "\n\nFinding min and max elements in a MyArray "
42       << "with std::ranges::minmax_element, which requires "
43       << "a forward range:\n";
44    auto [min, max] {std::ranges::minmax_element(strings)};
45    std::cout << "min and max elements of strings are: "
46       << *min << ", " << *max;
47
48    std::cout << "\n\nReversing a MyArray with std::ranges::reverse, "
49       << "which requires a bidirectional range:\n";
50    std::ranges::reverse(ints);
51    std::cout << "ints after reversing elements: ";
52    std::ranges::copy(ints, outputInt);
53    std::cout << "\n";
54 }
```

```
Displaying MyArrays with std::ranges::copy, which requires input iterators:
ints: 1 2 3 4 5 6 7 8
doubles: 1.1 2.2 3.3 4.4 5.5
strings: red orange yellow

Displaying a MyArray with a range-based for statement, which requires input
iterators:
1.1 2.2 3.3 4.4 5.5

Copying a MyArray with std::ranges::copy, which requires an input range and
output iterator:
strings2 after copying from strings: red orange yellow

Finding min and max elements in a MyArray with std::ranges::minmax_element,
which requires a forward range:
min and max elements of strings are: orange, yellow

Reversing a MyArray with std::ranges::reverse, which requires a
bidirectional range:
ints after reversing elements: 8 7 6 5 4 3 2 1
```

Илл. 15.12 | Использование MyArray с циклом for для диапазона значений и с алгоритмами стандартной библиотеки C++

Создание MyArray и отображение элементов с помощью std::ranges::copy

В строках 13–26 мы создаем три контейнера `MyArray` (строки 15, 19 и 24), используя CTAD для определения типов элементов и размера контейнера. В строках 17, 21 и 26 содержимое каждого контейнера отображается с помощью алгоритма `std::ranges::copy`. Первым аргументом этого алгоритма является `input_range`, требующий диапазона с итераторами ввода. Контейнеры `MyArray` совместимы с этим алгоритмом, потому что они предоставляют более мощные двунаправленные итераторы.

Отображение элементов MyArray в цикле for для диапазона значений

В строках 30–32 содержимое контейнера `doubles` отображается с помощью цикла `for` для диапазона значений. Это требует только итераторов ввода и поэтому работает с любыми итерируемыми объектами, включая объекты `MyArray`. Контейнер `doubles` не объявлен константным, поэтому используются итераторы `MyArray` для чтения и записи. Если у вас есть изменяемый объект, который вы хотите обрабатывать как константный, вы можете создать константное представление этого объекта, передав его в функцию `std::as_const` (C++17). В нашем примере цикл `for` для диапазона значений не изменяет элементы контейнера `double`, поэтому мы могли бы записать строку 30 таким образом:

```
for (auto& item : std::as_const(doubles)) {
```

В этом случае тип объекта `item` будет автоматически определен как константная ссылка на элемент типа `double`.

Копирование элементов MyArray с помощью алгоритма std::ranges::copy

В строке 36 создается новый контейнер `MyArray`, в который мы копируем элементы контейнера `strings`. В строке 37 используется алгоритм `std::ranges::copy` для копирования элементов `strings` в созданный контейнер `strings2`. Этот алгоритм использует итератор ввода для чтения каждого элемента контейнера `strings` и итератор вывода, указывающий позицию записи элемента в контейнер `strings2`. Неконстантные двунаправленные итераторы поддерживают как чтение (ввод), так и запись (вывод), поэтому один контейнер `MyArray` может быть скопирован алгоритмом `std::ranges::copy` в другой контейнер того же типа.

Поиск минимального и максимального элемента контейнера MyArray с помощью алгоритма std::ranges::minmax_element

В строке 44 используется алгоритм `std::ranges::minmax_element` для получения итераторов, указывающих на элементы, содержащие минимальные и максимальные значения контейнера. Этому алгоритму требуется диапазон `forward_range`, предоставляющий однонаправленные итераторы. Двунаправленные итераторы контейнера `MyArray` мощнее однонаправленных итераторов, поэтому алгоритм может работать с этим контейнером.

Изменяем порядок элементов на обратный с помощью алгоритма std::ranges::reverse

В строке 50 порядок элементов контейнера `ints` изменяется на обратный с помощью алгоритма `std::ranges::reverse`, которому требуется диапазон `bidirectional_range` с двунаправленными итераторами. Именно эти итераторы предоставляет `MyArray`, поэтому алгоритм может работать с этим контейнером.

Попытка использовать контейнер MyArray с алгоритмом std::ranges::sort

Итераторы контейнеров `MyArray` не поддерживают всю функциональность итераторов произвольного доступа, поэтому мы не можем передать контейнер алгоритмам, требующим эти итераторы. Чтобы доказать это, мы написали короткую программу, содержащую инструкции, которые создают контейнер `MyArray` с целочисленными элементами и пытаются отсортировать его с помощью алгоритма `std::ranges::sort`, которому требуются итераторы произвольного доступа:

```
MyArray integers{10, 2, 33, 4, 7, 1, 80};
std::ranges::sort(integers);
```

Компилятор Clang выдал сообщения об ошибках, показанные в выходных данных ниже. Мы выделили ключевые сообщения жирным шрифтом и добавили несколько пустых строк для удобства чтения. Сообщения ясно указывают, почему не компилируется код:

because 'MyArray<int, 7> &' does not satisfy 'random_access_range'

а также:

because 'iterator_t<MyArray<int, 7> &>' (aka 'Iterator<int>') does not satisfy 'random_access_iterator'

```
test.cpp:9:4: error: no matching function for call to object of type 'const
std::ranges::__sort_fn'
   std::ranges::sort(integers);
   ^~~~~~~~~~~~~~~~~

/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/bits/ranges_
algo.h:2032:7: note: candidate template ignored: constraints not satisfied
[with _Range = MyArray<int, 7> &, _Comp = std::ranges::less, _Proj =
std::identity]
      operator()(_Range&& __r, _Comp __comp = {}, _Proj __proj = {}) const
      ^

/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/bits/
ranges_algo.h:2028:14: note: because 'MyArray<int, 7> &' does not satisfy 'ran-
dom_access_range'
    template<random_access_range _Range,
             ^

/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/bits/
range_access.h:934:37: note: because 'iterator_t<MyArray<int, 7> &>' (aka 'It-
erator<int>') does not satisfy 'random_access_iterator'
= bidirectional_range<_Tp> && random_access_iterator<iterator_t<_Tp>>;
                                    ^

/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/bits/iter-
ator_concepts.h:614:10: note: because 'derived_from<__detail::__iter_con-
cept<Iterator<int> >, std::random_access_iterator_tag>' evaluated to false
      && derived_from<__detail::__iter_concept<_Iter>,
         ^
```

```
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/con-
cepts:67:28: note: because '__is_base_of(std::random_access_iterator_tag,
std::bidirectional_iterator_tag)' evaluated to false
    concept derived_from = __is_base_of(_Base, _Derived)
                           ^

/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/bits/rang-
es_algo.h:2019:7: note: candidate function template not viable: requires at
least 2 arguments, but 1 was provided
      operator()(_Iter __first, _Sent __last,
      ^
1 error generated.
```

15.10. Аргументы по умолчанию для параметров-типов шаблона

Для параметра-типа можно указать аргумент по умолчанию. Например, стандартный шаблон класса адаптера контейнера `stack` начинается с такого заголовка:

```
template <class T, class Container = deque<T>>
```

Это означает, что контейнер `stack` по умолчанию хранит свои элементы типа `T` в контейнере `deque`. Когда компилятор видит объявление:

```
stack<int> values;
```

он создает специализацию шаблона класса `stack` для типа `int` и затем использует ее для создания объекта с именем `values`. Элементы `stack` в этом случае хранятся в контейнере `deque<int>`.

Параметры-типы с аргументами по умолчанию должны быть последними (расположенными справа) в списке параметров-типов шаблона. При создании специализации шаблона с двумя или более аргументами по умолчанию, если пропущенный вами аргумент не является последним, все последующие также должны быть пропущены. В C++11 была добавлена возможность использовать аргументы по умолчанию для параметров-типов в шаблонах функций.

15.11. Шаблоны переменных

Вы создавали специализации шаблонов функций и шаблонов классов, определяя таким образом группы связанных функций и классов соответственно. В C++14 были добавлены **шаблоны переменных (variable templates)**, которые определяют группы связанных переменных. Вы уже встречали несколько предопределенных шаблонов переменных в коде на илл. 15.6, при рассмотрении признаков типов. В C++17 были добавлены удобные шаблоны переменных, позволяющие получать доступ к признакам типов через элемент данных `value`. Например, шаблон:

```
is_integral_v<T>
```

определяется в стандарте так:

```
template<class T>
inline constexpr bool is_arithmetic_v = is_arithmetic<T>::value;
```

Этот шаблон определяет переменную `inline bool`, вычисляемую как константа на этапе компиляции (обозначается ключевым словом `constexpr`). В C++17 была добавлена возможность объявления `inline`-переменных для лучшей поддержки заголовочных библиотек, которые могут быть включены в несколько файлов исходного кода (то есть единиц трансляции) одного приложения[1]. Когда вы определяете в заголовке обычную переменную, включение этого заголовка более одного раза в код приложения приводит к ошибкам множественного определения переменной. С другой стороны, идентичные определения `inline`-переменных допускаются в разных единицах трансляции одного приложения[2].

15.12. Вариативные шаблоны и выражения свертки

До C++11 каждый шаблон класса или шаблон функции имел фиксированное количество параметров. Для определения шаблона класса или функции с меняющимся количеством параметров надо было определять отдельный шаблон для каждого случая. Вариативные шаблоны, введенные в C++11, могут принимать произвольное количество аргументов. Они упрощают программирование с шаблонами: вы можете предоставить один вариативный шаблон функции вместо множества перегруженных шаблонов с разным количеством параметров.

15.12.1. Кортежи: вариативный шаблон класса tuple

Класс `tuple` (C++11, заголовок `<tuple>`) представляет обобщенный вариативный шаблон класса `pair`, который мы использовали в разделе 13.9. **Кортеж (tuple)** — это набор связанных значений, которые могут быть разных типов. На илл. 15.13 демонстрируется несколько возможностей этого класса.

```
1 // fig15_13.cpp
2 // Работа с кортежами
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <string>
6 #include <tuple>
7
```

Окончание ⇨

[1] Алекс Померанц (Alex Pomeranz), «6.8 — Global Constants and Inline Variables», 3 января 2020 г. Был доступен: 15 сентября 2023 г. https://www.learncpp.com/cpp-tutorial/global-constants-and-inline-variables/.

[2] «inline Specifier». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/language/inline.

```
 8 // Псевдоним для кортежа, представляющего информацию о товаре на складе
 9 using Part = std::tuple<int, std::string, int, double>;
10
11 // Возвращаем кортеж с информацией о товаре
12 Part getInventory(int partNumber) {
13    using namespace std::string_literals; // Строковые объектные литералы
14
15    switch (partNumber) {
16    case 1:
17       return {1, "Hammer"s, 32, 9.95}; // Возвращаем кортеж Part
18    case 2:
19       return {2, "Screwdriver"s, 106, 6.99}; // Возвращаем кортеж Part
20    default:
21       return {0, "INVALID PART"s, 0, 0.0}; // Возвращаем кортеж Part
22    }
23 }
24
25 int main() {
26    // Отображаем информацию о товарах на складе
27    for (int i{1}; i <= 2; ++i) {
28       // Распаковываем возвращенный кортеж в 4 переменные, типы которых
29       // автоматически определяются по значениям элементов кортежа
30       auto [partNumber, partName, quantity, price] {getInventory(i)};
31
32       std::cout << fmt::format("{}: {}, {}: {}, {}: {}, {}: {:.2f}\n",
33          "Part number", partNumber, "Tool", partName,
34          "Quantity", quantity, "Price", price);
35    }
36
37    std::cout << "\nAccessing a tuple's elements by index number:\n";
38    auto hammer{getInventory(1)};
39    std::cout << fmt::format("{}: {}, {}: {}, {}: {}, {}: {:.2f}\n",
40       "Part number", std::get<0>(hammer), "Tool", std::get<1>(hammer),
41       "Quantity", std::get<2>(hammer), "Price", std::get<3>(hammer));
42
43    std::cout << fmt::format("A Part tuple has {} elements\n",
44       std::tuple_size<Part>{}); // Получаем размер кортежа
45 }
```

```
Part number: 1, Tool: Hammer, Quantity: 32, Price: 9.95
Part number: 2, Tool: Screwdriver, Quantity: 106, Price: 6.99

Accessing a tuple's elements by index number:
Part number: 1, Tool: Hammer, Quantity: 32, Price: 9.95
A Part tuple has 4 elements
```

Илл. 15.13 | Работа с кортежами

Объявление `using` в строке 9 (рассмотрено в разделе 10.13) создает псевдоним типа с именем `Part` для кортежа, представляющего описание товаров на складе. В каждый элемент `Part` входят:

- переменная `int` — внутренний номер товара;
- переменная `string` — название товара;
- переменная `int` — количество товаров;
- переменная `double` — цена товара.

В объявлении кортежа каждый параметр-тип шаблона соответствует значению в одной и той же позиции кортежа. Количество элементов кортежа всегда соответствует количеству параметров-типов. Мы используем псевдоним `Part`, чтобы упростить остальную часть кода.

Упаковка кортежа

Создание объекта `tuple` называется **упаковкой кортежа (packing a tuple)**. В строках 12–23 определяется функция `getInventory`, получающая номер товара и возвращающая кортеж `Part`. Функция упаковывает кортеж `Part`, возвращая список инициализаторов (строки 17, 19 и 21), содержащий четыре элемента (`int`, `string`, `int` и `double`). Компилятор использует его для инициализации возвращаемого объекта `Part` `std::tuple`. Размер кортежа фиксируется при его создании.

Создание кортежа с помощью функции std::make_tuple

Вы также можете упаковать кортеж с помощью функции `make_tuple` из заголовка `<tuple>`, которая автоматически определяет параметры-типы кортежа по своим аргументам. Например, вы могли бы создать кортежи в строках 17, 19 и 21 таким вызовом функции `make_tuple`:

```
std::make_tuple(1, "Hammer"s, 32, 9.95)
```

Если использовать этот подход, возвращаемый тип функции `getInventory` можно указать как `auto`, чтобы он определялся по результату функции `make_tuple`.

Распаковка кортежа с помощью структурного связывания

Вы можете распаковать кортеж для получения доступа к его элементам, используя введенное в C++17 структурное связывание (раздел 14.4.5). В строке 30 элементы кортежа `Part` распаковываются в переменные, которые отображаются в строках 32–34.

Доступ к элементу кортежа по индексу с помощью get<index>

Шаблон класса `pair` содержит открытые элементы данных `first` и `second` для доступа к двум элементам `pair`. Каждый создаваемый кортеж может содержать любое количество элементов, и среди них не должно быть элементов с одинаковыми именами. Заголовок `<tuple>` предоставляет шаблон функции `get<`*индекс*`>(`*tupleObject*`)`, возвращающей ссылку на элемент *tupleObject* с указанным индексом. Первый элемент имеет индекс `0`. В строке 38 создается кортеж `hammer`, а в строках 39–41 мы получаем доступ к каждому элементу кортежа по индексу.

Использование get<type> для получения элемента кортежа по типу (C++14)

В C++14 была добавлена перегруженная версия `get`, возвращающая элемент кортежа определенного типа при условии, что в кортеже нет других элементов этого типа. Например, следующая инструкция получает элемент с типом `string` из кортежа `hammer`:

```
auto partName{get<std::string>(hammer)};
```

Однако инструкция:

```
auto partNumber{get<int>(hammer)};
```

приведет к ошибке компиляции, поскольку вызов неоднозначен. Кортеж `hammer` содержит два элемента типа `int`: номер товара и количество товаров.

Другие возможности кортежей

В следующей таблице показаны некоторые другие возможности шаблона класса `tuple`. Подробная информация о шаблоне класса `tuple` опубликована по этому адресу:

https://en.cppreference.com/w/cpp/utility/tuple/tuple

Здесь вы найдете описания остальных полезных инструментов программиста, определенных в заголовке `<tuple>`:

https://en.cppreference.com/w/cpp/utility/tuple

Возможности шаблона класса tuple	Описание
Сравнения	Кортежи, содержащие одинаковое количество элементов, можно сравнивать друг с другом операторами сравнения и равенства (при условии, что элементы кортежей поддерживают соответствующие операторы)
Конструктор по умолчанию	Позволяет создавать кортеж, в котором каждый элемент инициализирован значением. Значения простых типов инициализируются нулем или его эквивалентом. Объекты классов инициализируются их конструкторами по умолчанию
Копирующий конструктор	Копирует элементы кортежа в новый кортеж того же типа
Перемещающий конструктор	Перемещает элементы кортежа в новый кортеж того же типа
Копирующее присваивание	Позволяет использовать оператор присваивания (=) для копирования элементов кортежа (правый операнд) в кортеж того же типа (левый операнд)
Перемещающее присваивание	Позволяет использовать оператор присваивания (=) для перемещения элементов кортежа (правый операнд) в кортеж того же типа (левый операнд)

15.12.2. Вариативные шаблоны функций и основы выражений свертки (C++17)

Вариативные шаблоны функций позволяют определять функции, которые могут получать произвольное количество аргументов. В программе, показанной на илл. 15.14, такие шаблоны используются для получения суммы нескольких чисел — аргументов функции. В этом примере предполагается, что аргументы поддерживают оператор + и имеют один и тот же тип. Мы показываем два способа обработки вариативных параметров:

- используя рекурсию на этапе компиляции (которая требовалась до C++17);
- используя выражение свертки C++17, устраняющее необходимость в рекурсии.

В разделе 15.12.7 мы объясним, как убедиться в том, что все аргументы имеют один и тот же тип.

```
// fig15_14.cpp
// Вариативные шаблоны функций
#include <fmt/format.h>
#include <iostream>
#include <string>

// Функция базовой задачи для одного аргумента
template <typename T>
auto sum(T item) {
   return item;
}

// Рекурсивно прибавляем один или несколько аргументов
template <typename FirstItem, typename... RemainingItems>
auto sum(FirstItem first, RemainingItems... theRest) {
   return first + sum(theRest...); // Разворачиваем пакет параметров
                                   для следующего вызова
}

// Прибавляем один или несколько аргументов выражением свертки
template <typename FirstItem, typename... RemainingItems>
auto foldingSum(FirstItem first, RemainingItems... theRest) {
   return (first + ... + theRest); // Разворачиваем параметр
}

int main() {
   using namespace std::literals;

   std::cout << "Recursive variadic function template sum:"
      << fmt::format("\n{}\n{}\n{}\n{}\n\n",
         "sum(1): ", sum(1), "sum(1, 2): ", sum(1, 2),
         "sum(1, 2, 3): ", sum(1, 2, 3),
         "sum(\"s\"s, \"u\"s, \"m\"s): ", sum("s"s, "u"s, "m"s));

   std::cout << "Variadic function template foldingSum:"
```

Окончание ⇨

```
35        << fmt::format("\n{}{}\n{}{}\n{}{}\n{}{}\n",
36           "sum(1): ", foldingSum(1), "sum(1, 2): ", foldingSum(1, 2),
37           "sum(1, 2, 3): ", foldingSum(1, 2, 3),
38           "sum(\"s\"s, \"u\"s, \"m\"s): ",
39           foldingSum("s"s, "u"s, "m"s));
40 }
```

```
Recursive variadic function template sum:
sum(1): 1
sum(1, 2): 3
sum(1, 2, 3): 6
sum("s"s, "u"s, "m"s): sum

Variadic function template foldingSum:
sum(1): 1
sum(1, 2): 3
sum(1, 2, 3): 6
sum("s"s, "u"s, "m"s): sum
```

Илл. 15.14 | Вариативные шаблоны функций

Рекурсия на этапе компиляции

В строках 8–11 и 14–17 определяются перегруженные шаблоны функции `sum`, использующие рекурсию на этапе компиляции для обработки пакета параметров. Шаблон функции `sum` с одним параметром (строки 8–11) представляет базовый вариант рекурсии, в котором функция получает только один аргумент и возвращает его. Шаблон рекурсивной функции `sum` (строки 14–17) содержит два параметра-типа:

- `FirstItem` представляет первый аргумент функции;
- `RemainingItems` представляет все остальные аргументы, переданные функции.

Обозначение `typename...` предшествует пакету параметров шаблона, представляющему произвольное количество аргументов. В списке параметров функции (строка 15) вариативный параметр должен отображаться последним и обозначаться символами `...` после имени его типа (`RemainingItems`). Обратите внимание, что позиции символов `...` в заголовке шаблона и в списке параметров функции отличаются. Инструкция `return` прибавляет первый аргумент к результату рекурсивного вызова:

```
sum(theRest...)
```

Выражение `theRest...` является **разверткой пакета параметров (parameter-pack expansion)**: компилятор преобразует элементы пакета параметров в список, разделенный запятыми. Скоро мы поговорим об этом подробнее.

Вызов функции sum с одним аргументом

В строке 30 вызывается функция с одним аргументом:

```
sum(1)
```

В этом случае компилятор вызывает версию `sum` с одним аргументом (строки 8–11). Поскольку аргумент только один, результатом (суммой) будет его значение, в данном случае `1`.

Вызов функции sum с двумя аргументами

В строке 30 вызывается функция с двумя аргументами:

```
sum(1, 2)
```

В этом случае компилятор вызывает вариативную версию функции `sum` (строки 14–17), которая получает значение `1` в параметре `first` и значение `2` в пакете параметров `theRest`. Затем функция суммирует `1` и результат вызова функции `sum` с разверткой пакета параметров (`...`). Пакет параметров содержит только значение `2`, поэтому выполняется вызов `sum(2)`, то есть вызывается версия функции с одним аргументом, завершающая рекурсию. Итак, конечный результат равен `3` (`1 + 2`).

Хотя вариативная версия функции `sum` выглядит как традиционная рекурсия, компилятор генерирует отдельные специализации шаблона для каждого рекурсивного вызова. Поэтому вызов `sum(1, 2)` становится таким:

```
sum(1, sum(2))
```

точнее, таким:

```
sum<int, int>(1, sum<int>(2))
```

Компилятор имеет все значения, используемые в этой инструкции, поэтому он может уже на этапе компиляции выполнить вычисления и встроить результаты в программу в виде констант, чтобы избежать лишних вычислений при вызове функции во время выполнения[1].

Вызов функции sum с тремя аргументами

В строке 31 вызывается функция с тремя аргументами:

```
sum(1, 2, 3)
```

В этом случае компилятор снова вызывает вариативную версию функции `sum` (строки 14–17):

- В первом вызове параметр `first` получает `1`, а пакет параметров `theRest` получает `2` и `3`. Функция суммирует `1` и результат вызова `sum` с разверткой пакета параметров, выполняя вызов `sum(2, 3)`.
- В вызове `sum(2, 3)` параметр `first` получает `2`, а пакет параметров `theRest` получает `3`. Функция суммирует `2` и результат вызова функции `sum` с разверткой пакета параметров, поэтому выполняется вызов `sum(3)`.

[1] «Template Metaprogramming — Compile-Time Code Optimization». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Template_metaprogramming#Compile-time_code_optimization.

- Вызов `sum(3)` — это вызов версии функции с одним аргументом (базовой функции) с параметром `3`, который просто возвращает это число. На этом этапе вызов `sum(2, 3)` возвращает `2 + 3` (то есть `5`), а вызов `sum(1, 2, 3)` возвращает `1 + 5`, что является конечным результатом `6`.

Фактически первоначальный вызов становится таким:

```
sum(1, sum(2, sum(3)))
```

При этом компилятор знает значение последнего вложенного вызова `sum(3)` и может определить результаты остальных вызовов.

Вызов функции sum с тремя объектами string

Функция `sum` в строке 32 получает три строковых объектных литерала:

```
sum("s"s, "u"s, "m"s)
```

Напомним, что для объектов `string` оператор `+` выполняет конкатенацию строк, поэтому исходный вызов фактически становится таким:

```
sum("s"s, sum("u"s, sum("m"s)))
```

Результатом будет строка `"sum"`.

Выражения свертки (C++17)

Выражения свертки (fold expressions) дают вам удобный способ поочередной обработки бинарным оператором всех элементов в пакете параметров вариативного шаблона[1,2,3]. Выражения свертки часто применяются для того, чтобы преобразовать все значения в пакете параметров в одно значение. Эти выражения также могут применить операцию к каждому объекту в пакете параметров, например вызвать функцию или отобразить элементы пакета с помощью `cout` (как будет показано в разделе 15.12.6).

В строках 20–23 используется **бинарная левая свертка (binary left fold)** (строка 22):

```
(first + ... + theRest)
```

Она суммирует аргумент `first` и все аргументы в пакете параметров `theRest` (их может быть 0 или более). Выражения свертки заключаются в круглые скобки. Выражение бинарной левой свертки содержит два бинарных оператора, которые должны быть идентичными, в данном случае это операторы сложения (`+`).

---

[1] Джонатан Боккара (Jonathan Boccara), «C++ Fold Expressions 101», 12 марта 2021 г. Был доступен: 15 сентября 2023 г. https://www.fluentcpp.com/2021/03/12/cpp-fold-expressions/.

[2] Джонатан Боккара, «What C++ Fold Expressions Can Bring to Your Code», 19 марта 2021 г. Был доступен: 15 сентября 2023 г. https://www.fluentcpp.com/2021/03/19/what-c-fold-expressionscan-bring-to-your-code/.

[3] Джонатан Мюллер (Jonathan Muller), «Nifty Fold Expression Tricks», 5 мая 2020 г. Был доступен: 15 сентября 2023 г. https://www.foonathan.net/2020/05/fold-tricks/.

Аргумент слева от первого оператора — это начальное значение выражения. Символы ... разворачивают пакет параметров справа от второго оператора, отделяя бинарным оператором каждый параметр от следующего. Итак, в строке 37 вызов функции

```
foldingSum(1, 2, 3)
```

разворачивается бинарной левой сверткой в такое выражение:

```
((1 + 2) + 3)
```

Если пакет параметров пуст, значением выражения бинарной левой свертки является начальное значение, в данном случае `first`.

В C++ Core Guidelines рекомендуется использовать вариативные шаблоны функций для аргументов разных типов[1] и списки инициализаторов (`initializer_list`) для функций, получающих переменное количество аргументов одного типа[2]. Однако выражения свертки не работают со списками инициализаторов.

15.12.3. Типы выражений свертки

Далее мы используем следующие обозначения:

- *пакет* — пакет параметров;
- *оператор* — один из 32 бинарных операторов, которые можно использовать в выражениях свертки[3];
- *начальноеЗначение* — начальное значение бинарного выражения свертки. Например, сумма может начинаться с `0`, а произведение может начинаться с `1`.

Есть четыре типа выражений свертки. Круглые скобки обязательны в каждом из них:

- Выражение **унарной левой свертки (unary left fold)** содержит один бинарный оператор с разверткой пакета параметров (...) в качестве левого операнда и пакет параметров в качестве правого операнда:

  (... *оператор пакет*)

- Выражение **унарной правой свертки (unary right fold)** содержит один бинарный оператор с разверткой пакета параметров (...) в качестве правого операнда и пакет параметров в качестве левого операнда:

  (*пакет оператор* ...)

---

[1] C++ Core Guidelines, «T.100: Use Variadic Templates When You Need a Function That Takes a Variable Number of Arguments of a Variety of Types». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-variadic.

[2] C++ Core Guidelines, «T.103: Don't Use Variadic Templates for Homogeneous Argument Lists». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-variadic-not.

[3] «Fold Expression». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/language/fold.

- Выражение **бинарной левой свертки (binary left fold)** содержит два одинаковых бинарных оператора. Слева от первого оператора находится начальное значение, между операторами — развертка пакета параметров (...), а справа от второго оператора — пакет параметров:

  (*начальноеЗначение оператор ... оператор пакет*)

- Выражение **бинарной правой свертки (binary right fold)** содержит два одинаковых бинарных оператора. Справа от второго оператора находится начальное значение, между операторами — развертка пакета параметров (...), а слева от первого оператора — пакет параметров:

  (*пакет оператор ... оператор начальноеЗначение*)

15.12.4. Применение операторов в унарных выражениях свертки

Ключевое различие между выражениями левой и правой свертки заключается в том порядке, в каком они применяют свои операторы. Левые свертки группируются слева направо, а правые — справа налево. В зависимости от оператора группировка может давать различные результаты. На илл. 15.15 показаны операции унарной левой свертки и унарной правой свертки с операторами сложения (+) и вычитания (-):

- В строках 6–9 определяется шаблон `unaryLeftAdd`, использующий унарную левую свертку для сложения элементов в пакете параметров.
- В строках 11–14 определяется шаблон `unaryRightAdd`, использующий унарную правую свертку для сложения элементов в пакете параметров.
- В строках 16–19 определяется шаблон `unaryLeftSubtract`, использующий унарную левую свертку для вычитания элементов в пакете параметров.
- В строках 21–24 определяется шаблон `unaryRightSubtract`, использующий унарную правую свертку для вычитания элементов в пакете параметров.

```
1 // fig15_15.cpp
2 // Унарные выражения свертки
3 #include <fmt/format.h>
4 #include <iostream>
5
6 template <typename... Items>
7 auto unaryLeftAdd(Items... items) {
8    return (... + items); // Унарная левая свертка
9 }
10
11 template <typename... Items>
12 auto unaryRightAdd(Items... items) {
13    return (items + ...); // Унарная правая свертка
14 }
15
```

```
16 template <typename... Items>
17 auto unaryLeftSubtract(Items... items) {
18    return (... - items); // Унарная левая свертка
19 }
20
21 template <typename... Items>
22 auto unaryRightSubtract(Items... items) {
23    return (items - ...); // Унарная правая свертка
24 }
25
26 int main() {
27    std::cout << "Unary left and right fold with addition:"
28       << fmt::format("\n{}{}\n{}{}\n\n",
29          "unaryLeftAdd(1, 2, 3, 4): ", unaryLeftAdd(1, 2, 3, 4),
30          "unaryRightAdd(1, 2, 3, 4): ", unaryRightAdd(1, 2, 3, 4));
31
32    std::cout << "Unary left and right fold with subtraction:"
33       << fmt::format("\n{}{}\n{}{}\n",
34          "unaryLeftSubtract(1, 2, 3, 4): ",
35          unaryLeftSubtract(1, 2, 3, 4),
36          "unaryRightSubtract(1, 2, 3, 4): ",
37          unaryRightSubtract(1, 2, 3, 4));
38 }
```

```
Unary left and right fold with addition:
unaryLeftAdd(1, 2, 3, 4): 10
unaryRightAdd(1, 2, 3, 4): 10

Unary left and right fold with subtraction:
unaryLeftSubtract(1, 2, 3, 4): -8
unaryRightSubtract(1, 2, 3, 4): -2
```

Илл. 15.15 | Унарные выражения свертки

Унарные свертки для сложения

Рассмотрим вызов в строке 29:

```
unaryLeftAdd(1, 2, 3, 4)
```

Здесь пакет параметров содержит числа **1**, **2**, **3** и **4**. Эта функция выполняет унарную левую свертку с оператором **+**, вычисляющую выражение:

```
(((1 + 2) + 3) + 4)
```

Аналогично вызов в строке 30:

```
unaryRightAdd(1, 2, 3, 4)
```

выполняет унарную правую свертку, которая вычисляет выражение:

```
(1 + (2 + (3 + 4)))
```

Порядок, в котором применяется оператор **+**, не имеет значения, поэтому в данном случае оба выражения дают одинаковый результат **10**.

Унарные свертки для вычитания

В зависимости от порядка, в котором применяются некоторые операторы, результаты могут отличаться. Рассмотрим вызов в строке 35:

```
unaryLeftSubtract(1, 2, 3, 4)
```

Эта функция выполняет унарную левую свертку с оператором `-`, вычисляющую выражение

```
(((1 - 2) - 3) - 4)
```

Результатом этого выражения будет `-8`. С другой стороны, вызов в строке 37:

```
unaryRightSubtract(1, 2, 3, 4)
```

выполняет унарную правую свертку с оператором `-`, вычисляющую выражение:

```
(1 - (2 - (3 - 4)))
```

Результатом этого выражения будет `-2`.

Пакеты параметров в унарных выражениях свертки не должны быть пустыми

Унарные выражения свертки должны применяться только к пакетам параметров, содержащим хотя бы один элемент; иначе возникнет ошибка компиляции. Исключениями из этого правила являются только бинарные операторы `&&`, `||` и «запятая» (`,`):

- Операция `&&` с пустым пакетом параметров вычисляется как `true`.
- Операция `||` с пустым пакетом параметров вычисляется как `false`.
- Любая операция, выполняемая с пустым пакетом параметров и оператором «запятая», вычисляется как `void()`, то есть не выполняется никакая операция. Мы рассмотрим выражения свертки с этим оператором в разделе 15.12.6.

15.12.5. Применение операторов в бинарных выражениях свертки

Если возможен пустой пакет параметров, вы можете использовать бинарную левую свертку или бинарную правую свертку. Для каждой из них требуется начальное значение. Если пакет параметров пуст, результатом выражения свертки будет начальное значение. На илл. 15.16 показаны примеры бинарной левой и правой свертки для сложения и вычитания:

- В строках 6–9 определен шаблон `binaryLeftAdd`, использующий бинарную левую свертку для сложения элементов в пакете параметров с начальным значением `0`.
- В строках 11–14 определен шаблон `binaryRightAdd`, использующий бинарную правую свертку для сложения элементов в пакете параметров с начальным значением `0`.

- В строках 16–19 определен шаблон `binaryLeftSubtract`, использующий бинарную левую свертку для вычитания элементов в пакете параметров с начальным значением `0`.
- В строках 21–24 определен шаблон `binaryRightSubtract`, использующий бинарную правую свертку для вычитания элементов в пакете параметров с начальным значением `0`.

Еще раз обратите внимание, что группировка выражений свертки при сложении не имеет значения, но выражения свертки при вычитании дают разные результаты в зависимости от группировки.

```
 1 // fig15_16.cpp
 2 // Бинарные выражения свертки
 3 #include <fmt/format.h>
 4 #include <iostream>
 5
 6 template <typename... Items>
 7 auto binaryLeftAdd(Items... items) {
 8    return (0 + ... + items); // Бинарная левая свертка
 9 }
10
11 template <typename... Items>
12 auto binaryRightAdd(Items... items) {
13    return (items + ... + 0); // Бинарная правая свертка
14 }
15
16 template <typename... Items>
17 auto binaryLeftSubtract(Items... items) {
18    return (0 - ... - items); // Бинарная левая свертка
19 }
20
21 template <typename... Items>
22 auto binaryRightSubtract(Items... items) {
23    return (items - ... - 0); // Бинарная правая свертка
24 }
25
26 int main() {
27    std::cout << "Binary left and right fold with addition:"
28       << fmt::format("\n{}{}\n{}{}\n{}{}\n{}{}\n\n",
29          "binaryLeftAdd(): ", binaryLeftAdd(),
30          "binaryLeftAdd(1, 2, 3, 4): ", binaryLeftAdd(1, 2, 3, 4),
31          "binaryRightAdd(): ", binaryRightAdd(),
32          "binaryRightAdd(1, 2, 3, 4): ", binaryRightAdd(1, 2, 3, 4));
33
34    std::cout << "Binary left and right fold with subtraction:"
35       << fmt::format("\n{}{}\n{}{}\n{}{}\n{}{}\n",
36          "binaryLeftSubtract(): ", binaryLeftSubtract(),
37          "binaryLeftSubtract(1, 2, 3, 4): ",
38          binaryLeftSubtract(1, 2, 3, 4),
39          "binaryRightSubtract(): ", binaryRightSubtract(),
40          "binaryRightSubtract(1, 2, 3, 4): ",
```

Окончание ⇨

```
41          binaryRightSubtract(1, 2, 3, 4));
42 }
```

```
Binary left and right fold with addition:
binaryLeftAdd(): 0
binaryLeftAdd(1, 2, 3, 4): 10
binaryRightAdd(): 0
binaryRightAdd(1, 2, 3, 4): 10

Binary left and right fold with subtraction:
binaryLeftSubtract(): 0
binaryLeftSubtract(1, 2, 3, 4): -10
binaryRightSubtract(): 0
binaryRightSubtract(1, 2, 3, 4): -2
```

Илл. 15.16 | Бинарные выражения свертки

15.12.6. Оператор «запятая» для повторения операций

Оператор «запятая» вычисляет выражение слева от него, а затем вычисляет выражение справа от него. Значением выражения с оператором «запятая» является значение крайнего справа выражения. Вы можете комбинировать этот оператор с выражениями свертки, чтобы применять какую-либо операцию к каждому из элементов в пакете параметров. Функция `printItems` на илл. 15.17 отображает каждый элемент пакета параметров (`items`) в отдельной строке. В строке 7 выражение слева от оператора «запятая»:

```
(std::cout << items << "\n")
```

выполняется один раз для каждого из элементов в пакете параметров `items`.

```
 1 // fig15_17.cpp
 2 // Повторение действия с помощью оператора "запятая" и выражения свертки
 3 #include <iostream>
 4
 5 template <typename... Items>
 6 void printItems(Items... items) {
 7    ((std::cout << items << "\n"), ...); // Унарная правая свертка
 8 }
 9
10 int main() {
11    std::cout << "printItems(1, 2.2, \"hello\"):\n";
12    printItems(1, 2.2, "hello");
13 }
```

```
printItems(1, 2.2, "hello"):
1
2.2
hello
```

Илл. 15.17 | Повторение действия с помощью оператора «запятая» и выражения свертки

15.12.7. Ограничение элементов пакета параметров одним типом

При работе с выражениями свертки могут быть случаи, когда все элементы должны иметь одинаковый тип. Например, вариативный шаблон функции должен суммировать аргументы и возвращать результат такого же типа, как у аргументов. На илл. 15.18 показан пример проверки идентичности типов всех элементов пакета параметров с помощью предопределенного концепта `std::same_as`.

```
// fig15_18.cpp
// Ограничение пакета параметров вариативного шаблона функции,
// гарантирующее идентичность типов всех элементов
#include <concepts>
#include <iostream>
#include <string>

template <typename T, typename... Us>
concept SameTypeElements = (std::same_as<T, Us> && ...);

// Прибавляем один или несколько аргументов с помощью выражением свертки
template <typename FirstItem, typename... RemainingItems>
   requires SameTypeElements<FirstItem, RemainingItems...>
auto foldingSum(FirstItem first, RemainingItems... theRest) {
   return (first + ... + theRest); // Разворачиваем параметр
}

int main() {
   using namespace std::literals;

   foldingSum(1, 2, 3); // OK: все значения - числа int
   foldingSum("s"s, "u"s, "m"s); // OK: все значения - объекты std::string
   foldingSum(1, 2.2, "hello"s); // Ошибка: три разных типа
}
```

```
fig15_18.cpp:23:4: error: no matching function for call to 'foldingSum'
   foldingSum(1, 2.2, "hello"s); // error: three different types
   ^~~~~~~~~~

fig15_18.cpp:14:6: note: candidate template ignored: constraints not satisfied
[with FirstItem = int, RemainingItems = <double, std::basic_string<char>>]
auto foldingSum(FirstItem first, RemainingItems... theRest) {
     ^

fig15_18.cpp:13:13: note: because 'SameTypeElements<int, double,
std::basic_string<char> >' evaluated to false
requires SameTypeElements<FirstItem, RemainingItems...>
         ^

fig15_18.cpp:9:34: note: because 'std::same_as<int, double>' evaluated to
false
```

Окончание ⇨

```
concept SameTypeElements = (std::same_as<T, Us> && ...);
                                ^

/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/con-
cepts:63:19: note: because '__detail::__same_as<int, double>' evaluated to
false
      = __detail::__same_as<_Tp, _Up> && __detail::__same_as<_Up, _Tp>;
                  ^

/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/con-
cepts:57:27: note: because 'std::is_same_v<int, double>' evaluated to false
      concept __same_as = std::is_same_v<_Tp, _Up>;
                          ^

fig15_18.cpp:9:34: note: and 'std::same_as<int, std::basic_string<char> >'
evaluated to false
concept SameTypeElements = (std::same_as<T, Us> && ...);
                                ^

/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/con-
cepts:63:19: note: because '__detail::__same_as<int, std::basic_string<char>
>' evaluated to false
      = __detail::__same_as<_Tp, _Up> && __detail::__same_as<_Up, _Tp>;
                  ^

/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/con-
cepts:57:27: note: because 'std::is_same_v<int, std::basic_string<char> >'
evaluated to false
      concept __same_as = std::is_same_v<_Tp, _Up>;
                          ^

1 error generated.
```

Илл. 15.18 | Ограничение пакета параметров вариативного шаблона функции, гарантирующее идентичность типов всех элементов

Пользовательский концепт для вариативного шаблона

В строках 8–9 определяется пользовательский концепт `SameTypeElements` с двумя параметрами-типами:

- первый представляет единственный тип;
- второй — это **вариативный параметр-тип (variadic type parameter)**.

Ограничение в строке 9:

```
std::same_as<T, Us>
```

сравнивает тип `T` с каждым типом из пакета параметров `Us`. Это ограничение находится в выражении свертки, поэтому развертка пакета параметров (`...`) применяет `std::same_as<T, Us>` один раз к каждому типу из пакета параметров `Us`, сравнивая каждый тип в этом пакете с типом `T`. Если каждый элемент в пакете параметров

`Us` совпадает с типом `T`, ограничение вычисляется как `true`. В строке 13, в шаблоне функции `foldingSum`, наш концепт `SameTypeElements` применяется к параметрам-типам функции.

Вызов функции foldingSum

В функции `main` в строках 21 и 22 вызывается функция `foldingSum` с тремя целыми числами (`int`) и тремя строками (`string`) соответственно. Каждый вызов содержит три аргумента одного типа, поэтому эти инструкции будут успешно скомпилированы. Однако в строке 23 мы пытаемся вызвать функцию `foldingSum` с тремя аргументами разного типа: `int`, `double` и `string`. В этом вызове тип `T` концепта `SameTypeElements` — это `int`, а типы `double` и `string` помещены в пакет параметров `Us`. Конечно, типы `double` и `string` не являются типом `int`, поэтому каждое применение концепта `std::same_as<T, Us>` завершается неудачей. В выходных данных илл. 15.18 показаны сообщения об ошибках компилятора Clang C++. Мы выделили ключевые сообщения жирным шрифтом и добавили пустые строки для удобства чтения. Компилятор сообщает:

```
error: no matching function for call to 'foldingSum'
```

а также:

```
note: candidate template ignored: constraints not satisfied
```

Он также указывает в своих примечаниях на каждый неудачный вызов `std::same_as`, объясняя, что тип не совпадает с `int`:

```
note: because 'std::same_as<int, double>' evaluated to false
note: and 'std::same_as<int, std::basic_string<char> >' evaluated to false
```

15.13. Метапрограммирование на основе шаблонов

Как мы уже упоминали, современный C++ ориентирован на то, чтобы компилятор выполнял как можно больше работы на этапе компиляции, оптимизируя код для повышения быстродействия во время выполнения. Значительная часть этой работы по оптимизации выполняется с помощью **метапрограммирования на основе шаблонов (template metaprogramming)**, которое также называют **шаблонным метапрограммированием (template metaprogramming)**. Эта технология позволяет компилятору:

- оперировать типами;
- выполнять вычисления во время компиляции;
- генерировать оптимизированный код.

Концепты, перегрузка концептов и признаки типов, рассмотренные в этой главе, — все это важные средства метапрограммирования на основе шаблонов.

Эта технология метапрограммирования связана с рядом сложностей. Зато при ее правильном применении ваши программы будут работать быстрее, поэтому вам надо знать о тех возможностях, которые она дает.

Цель этого раздела — рассмотреть примеры метапрограммирования. Мы предлагаем вам в сносках множество ссылок на ресурсы для дальнейшего самостоятельного изучения. Мы покажем примеры, демонстрирующие:

- вычисление значений на этапе компиляции с помощью метафункций;
- вычисление значений на этапе компиляции с помощью `constexpr`-функций;
- использование признаков типов на этапе компиляции с помощью инструкций `constexpr if` для повышения быстродействия во время выполнения;
- операции с типами на этапе компиляции с помощью метафункций.

15.13.1. Шаблоны C++ обладают полнотой по Тьюрингу

Тодд Вельдхейзен доказал, что шаблоны C++ являются **полными по Тьюрингу (Turing complete)**[1,2], поэтому все то, что поддается вычислению, можно вычислить уже на этапе компиляции, применяя технологию метапрограммирования на основе шаблонов. В 1994 году, в эпоху начала стандартизации C++, Эрвин Унру одним из первых продемонстрировал возможность полезных вычислений на этапе компиляции. Он написал шаблонную метапрограмму, выводящую все простые числа меньше 30[3,4]. Программа не компилировалась, но результаты вычисления простых чисел выводились в сообщениях компилятора об ошибках! Здесь вы можете изучить исходный код этой программы (уже несовместимый с современным C++) и сообщения компилятора:

http://www.erwin-unruh.de/primorig.html

Вот несколько строк из этих сообщений. Специализации шаблона класса D, содержащие ряд простых чисел, выделены жирным шрифтом:

```
|    Type `enum{}' can't be converted to txpe `D<2>' ("primes.cpp",L2/C25).
|    Type `enum{}' can't be converted to txpe `D<3>' ("primes.cpp",L2/C25).
|    Type `enum{}' can't be converted to txpe `D<5>' ("primes.cpp",L2/C25).
|    Type `enum{}' can't be converted to txpe `D<7>' ("primes.cpp",L2/C25).
```

---

[1] Тодд Вельдхейзен (Todd Veldhuizen), «C++ Templates Are Turing Complete», 2003. Был доступен: 15 сентября 2023 г. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.3670.

[2] «Template Metaprogramming». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Template_metaprogramming.

[3] Эрвин Унру (Erwin Unruh), «Prime Number Computation», 1994. ANSI X3J16-94-0075/ISO WG21-4-62.

[4] Райнер Гримм (Rainer Grimm), «C++ Core Guidelines: Rules for Template Metaprogramming», 7 января 2019 г. Был доступен: 15 сентября 2023 г. https://www.modernescpp.com/index.php/c-core-guidelines-rules-for-template-metaprogramming. Мы благодарим Райнера Гримма за этот пост в блоге, указавший нам на историческую заметку об Эрвине Унру.

15.13.2. Вычисления на этапе компиляции

Цель вычислений на этапе компиляции — повышение быстродействия программы во время выполнения[1]. На илл. 15.19 приведены примеры метапрограммирования для вычисления факториалов на этапе компиляции. Сначала мы показываем рекурсивное решение с шаблонами, а затем, в качестве альтернативы, два решения на основе `constexpr`-функций: с итеративным и рекурсивным алгоритмами, рассмотренными в разделе 5.17. В C++ Core Guidelines рекомендуется выполнять вычисления на этапе компиляции именно на основе `constexpr`-функций[2], потому что при этом сохраняется традиционный синтаксис C++.

```
// fig15_19.cpp
// Вычисление факториалов на этапе компиляции
#include <iostream>

// Метафункция значения Factorial вычисляет факториалы рекурсивным методом
template <int N>
struct Factorial {
   static constexpr long long value{N * Factorial<N - 1>::value};
};

// Специализация Factorial для базовой задачи
template <>
struct Factorial<0> {
   static constexpr long long value{1}; // 0! = 1
};

// Вычисление рекурсивным методом на этапе компиляции
constexpr long long recursiveFactorial(int number) {
   if (number <= 1) {
      return 1; // Базовые задачи: 0! = 1 и 1! = 1
   }
   else { // Шаг рекурсии
      return number * recursiveFactorial(number - 1);
   }
}

// Вычисление итеративным методом на этапе компиляции
constexpr long long iterativeFactorial(int number) {
   long long factorial{1}; // Результат для 0! и 1!

   for (long long i{2}; i <= number; ++i) {
      factorial *= i;
   }

```

Продолжение ⇨

[1] «Template Metaprogramming». https://en.wikipedia.org/wiki/Template_metaprogramming

[2] C++ Core Guidelines. «T.123: Use constexpr Functions to Compute Values at Compile Time». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-fct.

```
    return factorial;
}

int main() {
    // "Вызов" метафункции значения требует создания специализации шаблона
    // и доступа к его статическому элементу данных value
    std::cout << "CALCULATING FACTORIALS AT COMPILE TIME "
       << "WITH A RECURSIVE METAFUNCTION"
       << "\nFactorial(0): " << Factorial<0>::value
       << "\nFactorial(1): " << Factorial<1>::value
       << "\nFactorial(2): " << Factorial<2>::value
       << "\nFactorial(3): " << Factorial<3>::value
       << "\nFactorial(4): " << Factorial<4>::value
       << "\nFactorial(5): " << Factorial<5>::value;

    // Вызываем рекурсивную функцию recursiveFactorial на этапе компиляции
    std::cout << "\n\nCALCULATING FACTORIALS AT COMPILE TIME "
       << "WITH A RECURSIVE CONSTEXPR FUNCTION"
       << "\nrecursiveFactorial(0): " << recursiveFactorial(0)
       << "\nrecursiveFactorial(1): " << recursiveFactorial(1)
       << "\nrecursiveFactorial(2): " << recursiveFactorial(2)
       << "\nrecursiveFactorial(3): " << recursiveFactorial(3)
       << "\nrecursiveFactorial(4): " << recursiveFactorial(4)
       << "\nrecursiveFactorial(5): " << recursiveFactorial(5);

    // Вызываем итеративную функцию iterativeFactorial на этапе компиляции
    std::cout << "\n\nCALCULATING FACTORIALS AT COMPILE TIME "
       << "WITH AN ITERATIVE CONSTEXPR FUNCTION"
       << "\niterativeFactorial(0): " << iterativeFactorial(0)
       << "\niterativeFactorial(1): " << iterativeFactorial(1)
       << "\niterativeFactorial(2): " << iterativeFactorial(2)
       << "\niterativeFactorial(3): " << iterativeFactorial(3)
       << "\niterativeFactorial(4): " << iterativeFactorial(4)
       << "\niterativeFactorial(5): " << iterativeFactorial(5) << "\n";
}
```

```
CALCULATING FACTORIALS AT COMPILE TIME WITH A RECURSIVE METAFUNCTION
Factorial(0): 1
Factorial(1): 1
Factorial(2): 2
Factorial(3): 6
Factorial(4): 24
Factorial(5): 120

CALCULATING FACTORIALS AT COMPILE TIME WITH A RECURSIVE CONSTEXPR FUNCTION
recursiveFactorial(0): 1
recursiveFactorial(1): 1
recursiveFactorial(2): 2
recursiveFactorial(3): 6
recursiveFactorial(4): 24
```

```
recursiveFactorial(5): 120

CALCULATING FACTORIALS AT COMPILE TIME WITH AN ITERATIVE CONSTEXPR FUNCTION
iterativeFactorial(0): 1
iterativeFactorial(1): 1
iterativeFactorial(2): 2
iterativeFactorial(3): 6
iterativeFactorial(4): 24
iterativeFactorial(5): 120
```

Илл. 15.19 | Вычисление факториалов на этапе компиляции

Метафункции

Вы можете выполнять вычисления на этапе компиляции с помощью метафункций. Как и все функции, с которыми вы уже работали, метафункции содержат аргументы и возвращают значения, но их синтаксис существенно отличается. Метафункции реализуются в виде шаблонов классов, обычно с помощью структур (`struct`). Аргументы метафункции — это элементы, используемые для специализации шаблона класса, а ее возвращаемое значение является открытым (`public`) элементом класса. Признаки типов, представленные в разделе 15.5, являются метафункциями.

Есть два вида метафункций:

- **Метафункция значения (value metafunction)** — это шаблон класса с элементом данных, объявленным как `public static constexpr`, обычно с именем `value`. Шаблон класса использует свои аргументы для вычисления `value` на этапе компиляции. Именно так мы реализовали вычисление факториалов.
- **Метафункция типа (type metafunction)** — это шаблон класса, который содержит вложенный элемент данных с именем `type`, обычно определяемым как псевдоним типа. Шаблон класса использует свои аргументы для операций с типом на этапе компиляции. В разделе 15.13.4 рассматриваются метафункции типа.

Имена элементов данных метафункций `value` и `type` — это соглашения[1], общепринятые и в стандартной библиотеке C++, и в метапрограммировании в целом.

Метафункция Factorial

Метафункция `Factorial` (строки 6–9) реализована в соответствии с рекурсивным определением факториала:

$$n! = n \cdot (n - 1)!$$

[1] Джоди Хагинс (Jody Hagins), «Template Metaprogramming: Type Traits (Part 1 of 2)», 22 сентября 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=tiAVWcjIF6o.

Метафункция `Factorial` содержит один параметр шаблона, не относящийся к типу: целочисленный параметр `N` (строка 6), представляющий аргумент метафункции. В соответствии с соглашением стандарта C++ для метафункций значения, в строке 8 определяется элемент данных с именем `value`, объявленный как `public static constexpr`. Факториалы быстро увеличиваются, поэтому мы объявили переменную с типом `long long`. Константа `value` инициализируется результатом выражения:

```
N * Factorial<N - 1>::value
```

Это выражение умножает `N` на результат «вызова» метафункции `Factorial` для `N - 1`.

«Вызов» метафункции

Мы «вызываем» метафункцию, создавая специализацию шаблона, что может показаться странным. Например, для вычисления факториала числа `3` подошла бы такая инструкция:

```
Factorial<3>::value
```

В ней значение `3` является аргументом шаблона, а `::value` — «возвращаемым значением». Эта инструкция заставляет компилятор создавать новый тип, представляющий факториал числа `3`. Ниже мы расскажем об этом подробнее.

Специализация метафункции Factorial для базовой задачи: Factorial<0>

Для рекурсивного вычисления базовой задачей является выражение `0!`, равное `1`. В рекурсивную метафункцию вы включаете базовую задачу с полной специализацией шаблона (строки 12–15). Такая специализация содержит строку `template <>` (строка 12), которая означает, что все аргументы шаблона должны быть явно указаны в угловых скобках после имени класса. Полная специализация шаблона `Factorial<0>` соответствует только вызову `Factorial` с аргументом `0`. В строке 14 значение элемента `value` устанавливается равным `1`[1].

Как компилятор выбирает метафункцию Factorial для аргумента 0

Когда компилятор обрабатывает вызов метафункции, например `Factorial<0>::value` (строка 43 в функции `main`), он должен определить, какой шаблон класса `Factorial` надо использовать. Аргументом шаблона является целое число `0`. Шаблон класса `Factorial` в строках 6–9 может соответствовать любому целочисленному значению, а полная специализация шаблона `Factorial<0>` в строках 12–15 соответствует только значению `0`. Компилятор всегда выбирает наиболее специализированный шаблон из нескольких подходящих. Итак, `Factorial<0>` соответ-

[1] В разделе 5.17 мы отмечали, что 0 и 1 являются базовыми для вычисления факториала. Для подобной реализации мы могли бы определить вторую полную специализацию шаблона с именем Factorial<1>. Но метафункция Factorial обрабатывает Factorial<1> как 1 * Factorial<0>.

ствует полной специализации шаблона, поэтому `Factorial<0>::value` возвращает значение 1.

Как компилятор выбирает метафункцию Factorial для аргумента 3

Теперь рассмотрим вызов метафункции `Factorial<3>::value` (строка 46 в функции `main`). Компилятор генерирует каждую специализацию, необходимую для получения конечного результата на этапе компиляции. Когда компилятор обрабатывает вызов `Factorial<3>::value`, он специализирует шаблон класса в строках 6–9 со значением `3` в качестве аргумента. Сначала строка 8 этой специализации становится такой:

```
3 * Factorial<2>::value
```

Компилятор видит, что для завершения определения `Factorial<3>` он должен снова специализировать шаблон для `Factorial<2>`. Теперь строка 8 этой специализации становится такой:

```
2 * Factorial<1>::value
```

После этого компилятор видит, что для завершения определения `Factorial<2>` он должен снова специализировать шаблон для `Factorial<1>`. И строка 8 этой специализации становится такой:

```
1 * Factorial<0>::value
```

Из полной специализации шаблона в строках 12–15 компилятор знает, что `Factorial<0>::value` возвращает `1`. Это завершает создание специализаций `Factorial` для `Factorial<3>`.

Теперь компилятор знает все, что нужно для завершения предыдущих специализаций:

- `Factorial<1>::value` содержит результат `1 * 1`, который равен 1.
- `Factorial<2>::value` содержит результат `2 * 1`, который равен 2.
- `Factorial<2>::value` содержит результат `3 * 2`, который равен 6.

Таким образом, компилятор может вставить в исполняемый код готовый результат — константу `6` вместо исходного вызова метафункции, устраняя необходимость в вычислениях во время выполнения. Каждый из вызовов метафункции `Factorial` в функции `main` (строки 43–48) компилятор обрабатывает таким же образом, что позволяет ему выполнять вычисления на этапе компиляции.

Функциональное программирование

Все то, что компилятор делает во время специализации шаблона, не изменяет значения переменных после их инициализации[1]. В метапрограммировании на основе шаблонов нет изменяемых переменных. Это отличительная черта функ-

[1] «Template Metaprogramming». https://en.wikipedia.org/wiki/Template_metaprogramming.

ционального программирования. Все значения, вычисленные в этом примере, являются константами, созданными на этапе компиляции.

Вычисления на этапе компиляции с помощью constexpr-функций

Как вы видели, использовать метафункции для вычисления значений на этапе компиляции сложнее, чем традиционные функции, выполняющие такие же вычисления во время выполнения программы. Но в C++ есть две возможности использования традиционных функций на этапе компиляции:

- функция, объявленная со спецификатором `constexpr`, может быть вызвана как на этапе компиляции, так и во время выполнения;
- в C++20 вы можете объявить функцию со спецификатором `consteval` (вместо `constexpr`), указывая, что она может быть вызвана только на этапе компиляции для получения константы.

Многие вычисления, выполняемые с помощью метафункций, проще реализовать традиционными функциями, объявленными со спецификатором `constexpr` или `consteval`. Такие функции широко применяются в **метапрограммировании на этапе компиляции (compile-time metaprogramming)**. Многие члены Комитета по стандартам C++ предпочитают метапрограммирование на основе `constexpr`, потому что метапрограммирование на основе шаблонов «сложно в использовании, плохо масштабируется и, по сути, эквивалентно изобретению нового языка в C++»[1]. Они предложили ряд усовершенствований синтаксиса `constexpr`, которые упростят многие аспекты метапрограммирования в будущих версиях C++.

В строках 18–25 и 28–36 определяются функции `recursiveFactorial` и `iterativeFactorial`, использующие алгоритмы из раздела 5.17 и традиционный синтаксис C++. Каждая функция объявлена как `constexpr`, поэтому компилятор может вычислить результат функции на этапе компиляции. В строках 53–58 демонстрируется рекурсивный метод, а в строках 63–68 — итеративный.

15.13.3. Условная компиляция в метапрограммировании на основе шаблонов и инструкции constexpr if

Другой способ увеличения объема работы на этапе компиляции для оптимизации программы во время выполнения — генерация кода, оптимально выполняемого при работе программы. В разделе 15.6.4 мы рассмотрели способ оптимизации выполнения функции `customDistance` при помощи перегрузки с концептами C++20. В этом примере компилятор выбирал правильный шаблон перегруженной функции на основе ограничений для типа, определяемого параметром шаблона. На илл. 15.20 показан измененный пример с шаблоном функции `customDistance`,

[1] Луи Дионн (Louis Dionne), «Metaprogramming By Design, Not By Accident», 18 июня 2017 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p0425r0.

демонстрирующим, как вы могли реализовать подобную функциональность до появления концептов. Эта функция содержит **статическую (compile-time)** инструкцию `if` (C++17), известную как `constexpr if`. Она использует `std::is_base_of_v` (сокращенный способ доступа к `value` признака типа `std::is_base_of`), чтобы сделать выбор между генерацией вычислений $O(1)$ или $O(n)$ в зависимости от того, каковы итераторы-аргументы функции:

- итераторы произвольного доступа (или производные итераторы категории непрерывных итераторов), подходящие для вычислений $O(1)$; или
- итераторы категорией ниже, подходящие для вычислений $O(n)$.

```
// fig15_20.cpp
// Реализация функции customDistance методом метапрограммирования на основе
   шаблонов
#include <array>
#include <iostream>
#include <iterator>
#include <list>
#include <ranges>
#include <type_traits>

// Вычисляем расстояние (количество элементов) между двумя итераторами ввода;
// требуются как минимум итераторы ввода
template <std::input_iterator Iterator>
auto customDistance(Iterator begin, Iterator end) {
   // Для итераторов произвольного доступа достаточно выполнить их вычитание
   if constexpr (std::is_base_of_v<std::random_access_iterator_tag,
      Iterator::iterator_category>) {

      std::cout << "customDistance with random-access iterators\n";
      return end - begin; // O(1) операций для итераторов произвольного доступа
   }
   else { // Для всех остальных итераторов
      std::cout << "customDistance with non-random-access iterators\n";
      std::ptrdiff_t count{0};

      // Перебираем элементы от begin до end, подсчитывая количество итераций;
      // O(n) операций для всех итераторов, кроме итераторов произвольного
         доступа
      for (auto iter{begin}; iter != end; ++iter) {
         ++count;
      }

      return count;
   }
}

int main() {
```

Окончание ⇨

```
36    const std::array ints1{1, 2, 3, 4, 5}; // Предоставляет итераторы
                                             произвольного доступа
37    const std::list ints2{1, 2, 3}; // Предоставляет двунаправленные итераторы
38
39    auto result1{customDistance(ints1.begin(), ints1.end())};
40    std::cout << "ints1 number of elements: " << result1 << "\n";
41    auto result2{customDistance(ints2.begin(), ints2.end())};
42    std::cout << "ints2 number of elements: " << result2 << "\n";
43 }
```

```
customDistance with random-access iterators
ints1 number of elements: 5
customDistance with non-random-access iterators
ints2 number of elements: 3
```

Илл. 15.20 | Реализация функции customDistance методом метапрограммирования на основе шаблонов

В строках 12–33 определяется шаблон функции `customDistance`. Поскольку для выполнения задачи требуются хотя бы итераторы ввода, в строке 12 параметр-тип `Iterator` ограничен концептом `std::input_iterator`. Хотя статическая инструкция `if` известна в стандарте C++ как `constexpr if`, она записана в коде как `if constexpr` (строка 15). Условие этой инструкции должно быть вычислено как значение `bool` на этапе компиляции.

В строках 15–16 используется `std::is_base_of_v` для сравнения двух типов:

- `std::random_access_iterator_tag`;
- `Iterator::iterator_category`.

Напомним, что итератор содержит вложенный тип с именем `iterator_category`, который определяет тип итератора, используя один из следующих «теговых» типов из заголовка `<iterator>`:

- `input_iterator_tag`
- `output_iterator_tag`
- `forward_iterator_tag`
- `bidirectional_iterator_tag`
- `random_access_iterator_tag`
- `contiguous_iterator_tag`

Все они реализованы в виде иерархии классов[1]. Выражение в строке 16:

```
Iterator::iterator_category
```

[1] «std::input_iterator_tag, std::output_iterator_tag, std::forward_iterator_tag, std::bidirectional_iterator_tag, std::random_access_iterator_tag, std::contiguous_iterator_tag». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/iterator/iterator_tags.

получает тег из типа итератора (`std::is_base_of_v`) и сравнивает его с первым типом аргумента:

```
std::random_access_iterator_tag
```

чтобы определить, являются ли они одним типом или `std::random_access_iterator_tag` является базовым классом `Iterator::iterator_category`. Результатом выражения будет `true` или `false`. Если результат `true`, компилятор инстанцирует код в теле `if constexpr` (строки 18–19), который оптимизирован для итераторов произвольного доступа, выполняющих вычисления $O(1)$. В противном случае компилятор инстанцирует тело `else` (строки 22–31), в котором используется подход $O(n)$, работающий со всеми другими типами итераторов. Реализации класса `std::array` в компиляторах g++ и clang++ используют указатели в качестве итераторов. Они не будут работать с этой реализацией `customDistance`, в то время как показанный в разделе 15.6.4 пример кода перегрузки с концептом работает с указателями в качестве итераторов.

15.13.4. Метафункции типа

Метафункции типа часто используются для добавления и удаления атрибутов типов на этапе компиляции. Это более совершенный метод метапрограммирования на основе шаблонов, в основном используемый разработчиками шаблонов классов библиотеки. На илл. 15.21 мы будем добавлять и удалять атрибуты типов с помощью пользовательских метафункций типа, подобных имеющимся в заголовке `<type_traits>`, чтобы показать их возможную реализацию. Мы будем использовать и предопределенные метафункции типа. Запомните, что стандартные признаки типов из `<type_traits>` всегда предпочтительнее ваших собственных.

```
 1 // fig15_21.cpp
 2 // Добавление и удаление атрибутов типа с помощью метафункций типа
 3 #include <fmt/format.h>
 4 #include <iostream>
 5 #include <type_traits>
 6
 7 // Добавляем const к типу T
 8 template <typename T>
 9 struct my_add_const {
10    using type = const T;
11 };
12
13 // Общий случай: тип не является указателем, поэтому присваиваем T вложенной
      переменной type
14 template <typename T>
15 struct my_remove_ptr {
16    using type = T;
17 };
18
19 // Частичная специализация шаблона: T - это тип указателя, поэтому удаляем *
20 template <typename T>
21 struct my_remove_ptr<T*> {
```

Окончание ⇨

```
    using type = T;
};

int main() {
    std::cout << fmt::format("{}\n{}{}\n{}{}\n\n",
        "ADD CONST TO A TYPE VIA A CUSTOM TYPE METAFUNCTION",
        "std::is_same_v<const int, my_add_const<int>::type>: ",
        std::is_same_v<const int, my_add_const<int>::type>,
        "std::is_same_v<int* const, my_add_const<int*>::type>: ",
        std::is_same_v<int* const, my_add_const<int*>::type>);

    std::cout << fmt::format("{}\n{}{}\n{}{}\n\n",
        "REMOVE POINTER FROM TYPES VIA A CUSTOM TYPE METAFUNCTION",
        "std::is_same_v<int, my_remove_ptr<int>::type>: ",
        std::is_same_v<int, my_remove_ptr<int>::type>,
        "std::is_same_v<int, my_remove_ptr<int*>::type>: ",
        std::is_same_v<int, my_remove_ptr<int*>::type>);

    std::cout << fmt::format("{}\n{}{}\n{}{}\n\n",
        "ADD REFERENCES TO TYPES USING STANDARD TYPE TRAITS",
        "std::is_same_v<int&, std::add_lvalue_reference<int>::type>: ",
        std::is_same_v<int&, std::add_lvalue_reference<int>::type>,
        "std::is_same_v<int&&, std::add_rvalue_reference<int>::type>: ",
        std::is_same_v<int&&, std::add_rvalue_reference<int>::type>);

    std::cout << fmt::format("{}\n{}{}\n{}{}\n{}{}\n",
        "REMOVE REFERENCES FROM TYPES USING STANDARD TYPE TRAITS",
        "std::is_same_v<int, std::remove_reference<int>::type>: ",
        std::is_same_v<int, std::remove_reference<int>::type>,
        "std::is_same_v<int, std::remove_reference<int&>::type>: ",
        std::is_same_v<int, std::remove_reference<int&>::type>,
        "std::is_same_v<int, std::remove_reference<int&&>::type>: ",
        std::is_same_v<int, std::remove_reference<int&&>::type>);
}
```

```
ADD CONST TO A TYPE VIA A CUSTOM TYPE METAFUNCTION
std::is_same_v<const int, my_add_const<int>::type>: true
std::is_same_v<int* const, my_add_const<int*>::type>: true

REMOVE POINTER FROM TYPES VIA A CUSTOM TYPE METAFUNCTION
std::is_same_v<int, my_remove_ptr<int>::type>: true
std::is_same_v<int, my_remove_ptr<int*>::type>: true

ADD REFERENCES TO TYPES USING STANDARD TYPE TRAITS
std::is_same_v<int&, std::add_lvalue_reference<int>::type>: true
std::is_same_v<int&&, std::add_rvalue_reference<int>::type>: true

REMOVE REFERENCES FROM TYPES USING STANDARD TYPE TRAITS
std::is_same_v<int, std::remove_reference<int>::type>: true
std::is_same_v<int, std::remove_reference<int&>::type>: true
std::is_same_v<int, std::remove_reference<int&&>::type>: true
```

Илл. 15.21 | Операции с типами на этапе компиляции

Добавление const к типу

В строках 8–11 реализована метафункция типа с именем `my_add_const`. Это упрощенная версия метафункции `std::add_const` из заголовка `<type_traits>`. По соглашению метафункция типа должна иметь открытый элемент данных с именем `type`. Когда создается специализация `my_add_const` с каким-либо типом, компилятор определяет псевдоним этого типа в строке 10, где типу предшествует спецификатор `const`.

В строках 26–31 функции `main` демонстрируется метафункция `my_add_const`. В строке 29 выражение

```
my_add_const<int>::type
```

создает специализацию шаблона с неконстантным типом `int`. Чтобы подтвердить, что спецификатор `const` был добавлен к типу `int`, мы используем `std::is_same_v` (сокращенный доступ к элементу `value` шаблона `std::is_same`), сравнивая тип `const int` с типом, возвращаемым предыдущим выражением. Они идентичны, поэтому результат равен `true`, как показано в выходных данных. Аналогично строка 31 показывает, что метафункция `my_add_const` тоже работает с типами указателей.

Удаление * из типа указателя

В строках 14–17 и 20–23 реализуется пользовательская метафункция типа с именем `my_remove_ptr`. Она имитирует метафункцию `std::remove_pointer` из заголовка `<type_traits>`, показывая, как такая метафункция может быть реализована. Для этого требуются два шаблона класса метафункции:

- Шаблон в строках 14–17 обрабатывает общий случай, в котором тип не является указателем. Псевдоним типа в строке 16 определяет элемент `type` как `T`, который может быть любого типа. Для типа, который не содержит * для обозначения указателя, эта метафункция просто возвращает тип, используемый для специализации шаблона.
- Шаблон класса метафункции в строках 20–23 соответствует только типам указателей. Для этой цели мы определяем **частичную специализацию шаблона (partial template specialization)**[1]. В отличие от полной специализации шаблона, которая начинается с заголовка `template<>`, частичная специализация шаблона также определяется заголовком `template` (строка 20) с одним или несколькими параметрами, которые указываются в угловых скобках после имени класса (строка 21). В этом случае частичная специализация заключается в том, что тип `T` должен быть указателем, о чем говорит обозначение `T*`. Чтобы удалить * из типа указателя, псевдоним типа в строке 22 определяет элемент `type` как `T`.

[1] Инбал Леви (Inbal Levi), «Exploration of C++20 Metaprogramming», 29 сентября 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=XgrjybKaIV8.

В строках 36 и 38 функции `main` демонстрируются пользовательские метафункции типа `my_remove_ptr`. Рассмотрим, как компилятор делает выбор для их вызова.

Создание специализации my_remove_ptr для типов, которые не являются указателями

Выражение в строке 36:

```
my_remove_ptr<int>::type
```

создает специализацию шаблона с типом `int`, который не является указателем. Компилятор должен решить, какое определение шаблона `my_remove_ptr` соответствует выражению. Поскольку в типе `int` отсутствует указатель `*`, это выражение соответствует только определению шаблона `my_remove_ptr` в строках 14–17, в котором элемент данных `type` получает тип аргумента `int`. Для проверки мы используем `std::is_same_v`, сравнивая тип `int` с типом, возвращаемым предыдущим выражением. Они идентичны, и результат равен `true`.

Создание специализации my_remove_ptr для типов, которые являются указателями

Выражение в строке 38:

```
my_remove_ptr<int*>::type
```

создает специализацию шаблона с типом указателя `int*`, который может соответствовать обоим определениям `my_remove_ptr`:

- первое определение может соответствовать любому типу;
- второе определение может соответствовать любому типу указателя.

Опять же, компилятор всегда выбирает наилучший соответствующий шаблон. В данном случае тип `int*` соответствует частичной специализации шаблона в строках 20–23:

- тип `int` в выражении соответствует строке 21;
- обозначение `*` в выражении совпадает с обозначением `*` в строке 21, отделяя его от типа `int`, — это то, что позволяет частичной специализации шаблона удалять указатель (`*`) из типа.

Подтверждая, что `*` был удален, мы используем `std::is_same_v`, сравнивая тип `int` с типом, возвращаемым предыдущим выражением. Они идентичны, и результат равен `true`.

Добавление ссылок lvalue и rvalue к типу

Предопределенные метафункции типа, изменяющие типы, работают примерно так же, как `my_add_const` и `my_remove_ptr`. В строках 43 и 45 функции `main` демонстрируются предопределенные метафункции типа `add_lvalue_reference` и `add_rvalue_reference`. В строке 43 тип `int` преобразуется в тип `int&` — ссылоч-

ный тип *lvalue*, а в строке 45 тип `int` преобразуется в тип `int&&` — ссылочный тип *rvalue*. Мы используем `std::is_same_v` для проверки результатов.

Удаление ссылок lvalue и rvalue из ссылочного типа

В строках 50, 52 и 54 проверяется предопределенная метафункция типа `std::remove_reference` из заголовка `<type_traits>`, удаляющая ссылки из типа. Мы создаем специализацию `std::remove_reference` с типами `int`, `int&` и `int&&` соответственно:

- типы, не являющиеся ссылочными (например, `int`), возвращаются как есть;
- ссылочные типы *lvalue* возвращаются без `&`;
- ссылочные типы *rvalue* возвращаются без `&&`.

Для проверки мы используем `std::is_same_v`, сравнивая тип `int` с типом, возвращаемым каждым выражением в строках 50, 52 и 54. Они идентичны, и результаты равны `true`.

15.14. Итоги

В этой главе мы продемонстрировали технологии обобщенного программирования и полиморфизма на этапе компиляции с помощью шаблонов и концептов. Мы использовали шаблоны классов для создания взаимосвязанных пользовательских классов, и объяснили различия между шаблоном, который вы написали, и разнообразными специализациями, которые компилятор создает из кода вашего шаблона.

Мы представили сокращенные шаблоны функций C++20 и шаблоны лямбда-выражений. Мы использовали концепты C++20 для ограничения параметров шаблонов и для перегрузки шаблонов функций на основе их требований к типам. Далее мы рассмотрели признаки типов и показали их соотношение с концептами C++20. Мы создали пользовательский концепт и протестировали его на этапе компиляции с помощью объявления `static_assert`.

Мы рассмотрели создание ограниченного концептами алгоритма и обработку этим алгоритмом объектов нескольких классов контейнеров стандартной библиотеки C++. Затем мы доработали наш класс `myArray`, получив пользовательский шаблон класса контейнера с пользовательскими итераторами. Мы показали, что созданные нами итераторы позволяют алгоритмам стандартной библиотеки C++ обрабатывать элементы контейнера класса `myArray`. Мы использовали нетипизированные параметры шаблонов для передачи шаблонам констант, созданных на этапе компиляции, и рассмотрели аргументы шаблона по умолчанию.

Вы узнали, что вариативные шаблоны могут получать любое количество параметров. Сначала мы рассмотрели вариативный шаблон класса `tuple` стандартной библиотеки, а затем вариативные шаблоны функций. Мы также

показали, как применять бинарные операторы к вариативным параметров в выражениях свертки.

Многие методы, рассмотренные в этой главе, открывают перед программистами новые возможности метапрограммирования на этапе компиляции. В заключение мы кратко обсудили несколько принципиально разных концепций метапрограммирования, включая вычисления на этапе компиляции с помощью метафункций значения, условную компиляцию с помощью инструкций `constexpr if` и изменение типов с помощью метафункций типа.

В следующей главе мы представим вам модули, входящие в «большую четверку» языковых средств, добавленных в C++20 (наряду с диапазонами, концептами и корутинами). Модули — это совершенно новый способ структурирования кода, позволяющий надежно контролировать все объявления, доступные клиентскому коду, и инкапсулировать детали реализации.

16

Модули (C++20): технология разработки больших программ

В этой главе вы:

- узнаете о преимуществах модулей, особенно в больших программных системах;
- узнаете, как модули поддерживают инкапсуляцию;
- научитесь импортировать заголовки стандартной библиотеки в качестве блоков модуля;
- научитесь определять основной интерфейс модуля;
- научитесь экспортировать объявления из модулей, делая их доступными для других единиц трансляции;
- научитесь импортировать модули и использовать их экспортированные элементы;
- научитесь отделять интерфейс модуля от его реализации, размещая реализацию во фрагменте `:private` или в единице трансляции модуля;
- узнаете об ошибках компиляции, возникающих при попытке использовать неэкспортированные элементы модуля;
- научитесь использовать разделы модуля для организации логических компонентов;
- научитесь разделять модули на «подмодули», чтобы разработчики клиентского кода могли выбирать только те части библиотеки, которые они хотят использовать;
- узнаете разницу между видимостью и доступностью;
- узнаете, как модули могут сократить размеры единиц трансляции и время компиляции.

16.1. Введение

Модули относятся к «большой четверке» новинок C++20 (вместе с диапазонами, концептами и корутинами). Они дают вам новый способ организации кода, позволяя точно контролировать, какие объявления вы открываете для клиентского кода, и инкапсулировать детали реализации[1]. Каждый модуль — это группа взаимосвязанных объявлений и определений с уникальным именем и четко определенным интерфейсом, которым может пользоваться клиентский код.

В этой главе представлены практические примеры кода с модулями. Модули помогают разработчикам работать продуктивнее, особенно при создании, поддержке и развитии больших программных систем[2]. Благодаря модулям, такие системы можно делать более масштабируемыми[3]. Создатель C++ Бьерн Страуструп говорит: «Модули дают языку C++ историческую возможность улучшить чистоту кода и время компиляции (перенося C++ в XXI век)»[4].

Даже в небольших системах модули могут давать немедленные преимущества. Например, вы можете заменить директивы препроцессора стандартной библиотеки C++ `#include` инструкциями `import`. Это исключает многократную обработку одних и тех же файлов, включенных в вашу программу директивами `#include`, поскольку и импортированные заголовки в стиле модулей (раздел 16.4), и сами модули компилируются только один раз и используются повторно, когда вы импортируете их в программу.

Краткая история модулей

Модули C++ было сложно реализовать, на это ушло много лет. Первые проекты были предложены в начале 2000-х, но тогда Комитет по стандартам C++ был сосредоточен на других целях, которые в итоге привели к C++11. В 2014 году был предложен проект «Модульная система для C++», основанный на работе Габриэля Дос Рейса и Бьерна Страуструпа в Техасском университете A&M[5,6]. Комитет по

---

[1] Давид Вандевурд (Daveed Vandevoorde), «Modules in C++ (Revision 6)», 11 января 2012 г. https://wg21.link/n3347.

[2] Габриэль Дос Рейс (Gabriel Dos Reis), «Programming in the Large with C++20: Meeting C++ 2020 Keynote», 11 декабря 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=j4du4LNsLiI.

[3] Васил Василев (Vassil Vassilev), Дэвид Ланге (David Lange), Малик Шахзад Музаффар (Malik Shahzad Muzaffar), Мирчо Родозов (Mircho Rodozov), Оксана Шадура (Oksana Shadura) и Александр Пенев (Alexander Penev), «C++ Modules in ROOT and Beyond», 25 августа 2020 г. Был доступен: 15 сентября 2023 г. https://arxiv.org/pdf/2004.06507.pdf.

[4] Бьерн Страуструп, «Modules and Macros», 11 февраля 2018 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p0955r0.

[5] Бьерн Страуструп (Bjarne Stroustrup), «Thriving in a Crowded and Changing World: C++ 2006–2020 — Section 9.3.1 Modules», 12 июня 2020 г. Был доступен: 15 сентября 2023 г. https://www.stroustrup.com/hopl20-main-p5-p-bfc9cd4--final.pdf.

[6] Габриэль Дос Рейс (Gabriel Dos Reis), Марк Холл (Mark Hall) и Гор Нишанов (Gor Nishanov), «A Module System for C++». Был доступен: 15 сентября 2023 г. https://wg21.link/n4047.

стандартам C++ сформировал исследовательскую группу по модулям, разработавшую их техническую спецификацию[1]. Последующие обсуждения в Комитете в основном были сосредоточены на технических компромиссах и деталях спецификации модулей в C++20[2].

Поддержка модулей компилятором

На момент написания этой книги три наших компилятора еще не полностью поддерживали модули. Мы протестировали все примеры на каждом компиляторе и приводим в книге необходимые команды компиляции для каждого примера, отмечая те случаи, когда компилятор не поддерживает какую-либо операцию[3].

Контейнеры Docker для g++ 11.2 и clang++ 13

Для проверки примеров кода в последних версиях g++ и clang++ мы использовали бесплатные контейнеры Docker с сайта https://hub.docker.com:

- Для g++ мы взяли последнюю версию официального контейнера GNU GCC (11.2):

```
docker pull gcc:latest
```

- В настоящее время разработчики LLVM/Clang не предоставляют свой официальный контейнер Docker, но многие готовые к работе контейнеры есть на сайте hub.docker.com. Мы взяли последнюю и чаще всего загружаемую версию, содержащую clang++ версии 13[4]:

```
docker pull silkeh/clang:13
```

Если вы не умеете запускать контейнеры Docker, ознакомьтесь с разделом «Перед началом работы», а также с обзором Docker и инструкциями на сайте:

https://docs.docker.com/get-started/overview/

Компиляция и компоновка в языке C++ до выпуска C++20

В разделе 16.2 мы обсудим традиционный для C++ процесс компиляции и компоновки, отметив связанные с ним проблемы.

---

[1] Габриэль Дос Рейс (Gabriel Dos Reis) (ред.), «Working Draft, Extensions to C++ for Modules», 29 января 2018 г. Был доступен: 15 сентября 2023 г. https://wg21.link/n4720.

[2] Бьерн Страуструп (Bjarne Stroustrup), «Thriving in a Crowded and Changing World: C++ 2006–2020 – Section 9.3.1 Modules», 12 июня 2020 г. Был доступен: 15 сентября 2023 г. https://www.stroustrup.com/hopl20-main-p5-p-bfc9cd4--final.pdf.

[3] Шаги компиляции могут измениться. Если у вас возникнут проблемы, напишите нам по адресу deitel@deitel.com. Мы вскоре ответим и опубликуем обновления на сайте https://deitel.com/cpp20fp.

[4] Версия компилятора clang++ в Xcode не имеет такого количества реализованных функций C++20, как версия непосредственно от команды LLVM/Clang. Кроме того, на момент написания этой книги использование «latest» вместо «13» в команде docker pull выдает контейнер Docker с версией clang++ 12, а не 13.

Преимущества и цели модулей

В разделе 16.3 мы рассмотрим различные преимущества модулей, в том числе те, которые решают проблемы и улучшают компиляцию по сравнению с традиционным процессом (до C++20).

Будущее модулей и инструментов для работы с ними

В разделе 16.12 рассматриваются некоторые возможности модулей в C++23, которые на момент написания книги находятся в стадии разработки. Там же предоставляются ссылки на дополнительную информацию.

Глоссарий по модулям

В конце главы вы найдете приложения, содержащие список видеоуроков, статей, рефератов и документации, на которые мы ссылались при написании этой главы, а также глоссарий по модулям с ключевыми терминами и определениями. Все это поможет вам при желании самостоятельно углубиться в эту тему.

16.2. Компиляция и компоновка до C++20

C++ всегда имел модульную архитектуру управления кодом, основанную на комбинации заголовочных файлов и файлов исходного кода:

- Следуя имеющимся в исходном коде директивам препроцессора, препроцессор C++ выполняет в каждом файле исходного кода операции замены фрагментов текста и другую подготовку к компиляции. Обработанный препроцессором файл исходного кода называется **единицей трансляции (translation unit)**[1].
- Компилятор преобразует каждую единицу трансляции в файл объектного кода.
- Компоновщик объединяет файлы объектного кода с объектными файлами библиотеки (например, с файлами стандартной библиотеки C++) для создания готового исполняемого файла программы.

Этот подход сохраняется с 1970-х годов, и C++ унаследовал его от языка C[2].

Заголовочный файл/файл исходного кода — проблемы с текущей моделью

Препроцессор — это всего лишь механизм замены фрагментов текста, он не знает языка C++. В качестве аргументов для введения модулей C++20 создатель C++ Бьерн Страуструп называет три проблемы препроцессора[3]:

---

[1] «Translation Unit (Programming)». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Translation_unit_(programming).

[2] Габриэль Дос Рейс (Gabriel Dos Reis), «Programming with C++ Modules: Guide for the Working Software Developer», 5 октября 2019 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=tjSuKOz5HK4.

[3] Бьерн Страуструп (Bjarne Stroustrup), «Thriving in a Crowded and Changing World».

- Содержимое каждого заголовка может влиять на любой следующий заголовок, поэтому порядок включения заголовков важен и может вызвать неявные ошибки.
- Конкретный проект может содержать отличающиеся объявления в нескольких единицах трансляции. Компилятор и компоновщик не всегда сообщают о таких проблемах.
- Повторная обработка одного и того же заголовка происходит медленно, особенно в больших проектах, где один заголовок может включаться в разные места кода десятки или сотни раз. В каждую отдельную единицу трансляции препроцессор вставит содержимое заголовка, что, возможно, приведет к повторной компиляции одного и того же кода во многих единицах трансляции. Устранение этой проблемы путем импорта заголовков в виде **заголовочных единиц (header units)** трансляции (раздел 16.4) может значительно снизить время компиляции в больших проектах.

Существуют и другие проблемы препроцессора[1]:

- Определения в заголовках могут нарушать правило одного определения C++. Согласно этому правилу, «ни одна единица трансляции не должна содержать более одного определения любой переменной, функции, типа класса, типа перечисления, шаблона, аргумента по умолчанию для параметра (для функции в данной области видимости) или аргумента шаблона по умолчанию»[2].
- Заголовки не обеспечивают инкапсуляции. Все, что содержится в заголовке, доступно той единице трансляции, в которую он включен.
- Случайные циклические зависимости между заголовками вызывают ошибки компиляции и другие проблемы[3].
- В заголовках часто определены макросы — директивы препроцессора `#define`, создающие константы, представленные в виде символов (например, `#define SIZE 10`), а также операции, подобные функциям (например, `#define SQUARE(x) ((x) * (x))`)[4]. Макросы обрабатываются препроцессором методом замены текстовых фрагментов. Компилятор не может проверить их синтаксис и не может сообщить об ошибках множественного определения, если в нескольких заголовках определено одно и то же имя макроса.

---

[1] Брайс Адельштейн Лельбах (Bryce Adelstein Lelbach). «Modules Are Coming», 1 мая 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=yee9i2rUF3s.

[2] Стандарт C++20, «6 Basics — 6.3 One-Definition Rule». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/basic.def.odr.

[3] «Circular Dependency». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Circular_dependency.

[4] Обозначение заглавными буквами констант препроцессора (например, SIZE) и макросов (например, SQUARE) — это соглашение, которое предпочитают многие программисты.

16.3. Преимущества и предназначение модулей

Некоторые преимущества использования модулей[1, 2, 3, 4, 5]:

- Улучшение организации и разделения на компоненты больших баз исходного кода.
- Уменьшение размеров единиц трансляции.
- Сокращение времени компиляции[6].
- Отсутствие повторной обработки заголовков. Скомпилированный модуль не требуется повторно обрабатывать для каждого файла исходного кода, который его использует.
- Отсутствие проблем с порядком включения заголовков.
- Отсутствие многих директив препроцессора, которые могут приводить к неявным ошибкам.
- Отсутствие нарушений правила одного определения.

Недостатки модулей

У модулей есть и недостатки[7, 8]:

- На момент написания этой книги поддержка модулей в большинстве компиляторов C++ была неполной.
- Существующие базы кода необходимо модифицировать, чтобы в полной мере использовать преимущества модулей.
- Модули не решают свойственные языку C++ проблемы неудобной упаковки и распространения программного обеспечения, в отличие от менеджеров пакетов в некоторых других популярных языках.

---

1 Корентен Жабо (Corentin Jabot), «What Do We Want from a Modularized Standard Library?», 16 мая 2020 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p2172r0.

2 Габриэль Дос Рейс (Gabriel Dos Reis) и Павел Кертис (Pavel Curtis), «Modules, Componentization, and Transition», 5 октября 2015 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p0141r0.

3 Даниэла Энгерт (Daniela Engert), «Modules: The Beginner's Guide», 2 мая 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=Kqo-jIq4V3I.

4 Райнер Гримм (Rainer Grimm), «C++20: The Advantages of Modules», 10 мая 2020 г. Был доступен: 15 сентября 2023 г. https://www.modernescpp.com/index.php/cpp20-modules.

5 Дмитрий Гузеев, «A Few Words on C++ Modules», 8 января 2018 г. Был доступен: 15 сентября 2023 г. https://medium.com/@dmitrygz/brief-article-on-c-modules-f58287a6c64.

6 Рене Ривера (Rene Rivera), «Are Modules Fast? (Revision 1)», 6 марта 2019 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p1441r1.

7 Стив Дауни (Steve Downey), «Writing a C++20 Module», 5 июля 2021 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=AO4piAqV9mg.

8 «C++ Modules Might Be Dead-on-Arrival», 27 января 2019 г. Был доступен: 15 сентября 2023 г. https://vector-of-bool.github.io/2019/01/27/modules-doa.html.

- Скомпилированные модули имеют специфические для компилятора аспекты, поэтому их нельзя портировать между компиляторами и платформами. Вам все равно надо распространять модули в виде исходного кода.
- Внедрение модулей будет медленным, поскольку организации внимательно изучают их возможности, ищут подходы для структурирования кода и потенциальной модификации существующего кода, а тем временем ждут информации об опыте других.
- На данный момент существует очень мало рекомендаций и руководств по модулям. Например, в C++ Core Guidelines на момент написания книги еще не было информации о модулях.

16.4. Пример перехода на модули: заголовочные единицы трансляции

Одна из целей модулей — устранить необходимость в препроцессоре. Существует большое количество библиотек, написанных до появления модулей. Большинство библиотек сегодня предоставляется в виде:

- заголовочных библиотек;
- комбинаций заголовков и файлов исходного кода;
- комбинаций заголовков и файлов объектного кода, специфичных для конкретной платформы.

Разработчикам библиотек потребуется время для их модификации. Некоторые библиотеки, скорее всего, никогда не будут поддерживать модули. В качестве переходного шага от препроцессора к модулям вы можете импортировать большинство существующих заголовков стандартной библиотеки C++[1], как показано в строке 3 на илл. 16.1. При этом заголовок рассматривается как заголовочная единица трансляции, сокращенно — **заголовочная единица (header unit)**.

```
1 // fig16_01.cpp
2 // Импорт заголовка стандартной библиотеки в виде заголовочной единицы
3 import <iostream>; // Вместо #include <iostream>
4
5 int main() {
6    std::cout << "Welcome to C++20 Modules!\n";
7 }
```

```
Welcome to C++20 Modules!
```

Илл. 16.1 | Импорт заголовка стандартной библиотеки в виде заголовочной единицы трансляции

[1] Стандарт C++20, «10 Modules — 10.3 Import Declaration». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/module.import.

Чем заголовочные единицы отличаются от заголовочных файлов

Вместо включения содержимого заголовка в файл исходного кода препроцессором компилятор обрабатывает заголовок как единицу трансляции, компилируя его и создавая информацию для обработки заголовка как модуля. В крупномасштабных системах это повышает быстродействие во время компиляции, поскольку заголовок компилируется только один раз вместо добавления его содержимого в каждую единицу трансляции, в которую он включен[1]. Заголовочные единицы аналогичны предварительно скомпилированным заголовкам в некоторых реализациях C++[2].

В отличие от директивы `#include`, порядок импорта не имеет значения. Например, последовательность инструкций[3]:

```
import SomeModule;
import SomeOtherModule;
```

дает тот же результат, что и обратная последовательность:

```
import SomeOtherModule;
import SomeModule;
```

Объявления заголовочной единицы доступны импортирующей единице трансляции, поскольку заголовочные единицы неявно «экспортируют» свое содержимое[4]. В разделе 16.6 вы увидите, что пользовательские модули могут указывать, какие объявления надо экспортировать (то есть делать доступными) для использования в других единицах трансляции. Это дает вам контроль над интерфейсом каждого модуля. В отличие от директивы `#include`, инструкция `import` не добавляет исходный код в импортирующую единицу трансляции. Кроме того, директивы препроцессора в единице трансляции, расположенные перед импортом заголовочной единицы, не влияют на содержимое заголовочной единицы[5].

Импортируйте все заголовки в виде заголовочных единиц (по возможности)

Как правило, вы должны импортировать все заголовки как заголовочные единицы[6]. К сожалению, не все заголовки могут быть импортированы. Если импорт

---

1 Габриэль Дос Рейс (Gabriel Dos Reis), «Programming with C++ Modules: Guide for the Working Software Developer».

2 Кэмерон Да Камара (Cameron DaCamara), «Practical C++20 Modules and the Future of Tooling Around C++ Modules», 4 мая 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=ow2zV0Udd9M.

3 Бьерн Страуструп (Bjarne Stroustrup), «Thriving in a Crowded and Changing World».

4 C++20 Standard: 10 Modules — 10.3 Import Declaration». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/module.import.

5 «Understanding C++ Modules: Part 3: Linkage and Fragments», 7 октября 2019 г. Был доступен: 15 сентября 2023 г. https://vector-of-bool.github.io/2019/10/07/modules-3.html.

6 Даниэла Энгерт (Daniela Engert), «Modules: The Beginner's Guide», 2 мая 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=Kqo-jIq4V3I.

заголовка в виде заголовочной единицы приводит к ошибкам компиляции, используйте традиционную директиву `#include`. Так бывает, если заголовок зависит от макросов `#define` препроцессора, которые связаны с состоянием препроцессора.

Компиляция программы с заголовочными единицами в Microsoft Visual Studio

Для компиляции любого примера из этой главы, в котором используются заголовочные единицы, убедитесь, что ваш проект Visual Studio настроен правильно:

1. Щелкните правой кнопкой мыши на название проекта в `Solution Explorer` и выберите `Properties`.
2. В диалоговом окне `Property Pages` выберите `All Configurations` из раскрывающегося списка конфигурации.
3. В левой колонке, в разделе `Configuration Properties > C/C++ > Language`, установите для `C++ Language Standard` значение `ISO C++20 Standard (/std:c++20)`[1].
4. В левой колонке, в разделе `Configuration Properties > C/C++ > All Options`, установите для параметра `Scan Sources for Module Dependencies` значение `Yes`.
5. Нажмите `Apply`, затем `OK`.

После такой настройки добавьте `fig16_01.cpp` в папку с исходными файлами вашего проекта, скомпилируйте и запустите проект.

Компиляция программы с заголовочными единицами в g++

В g++ вы должны сначала скомпилировать каждый заголовок, который будет импортирован как заголовочная единица[2, 3, 4]. Следующая команда компилирует заголовок стандартной библиотеки `<iostream>` как заголовочную единицу:

```
g++ -fmodules-ts -x c++-system-header iostream
```

- Флаг компилятора `-fmodules-ts` в настоящее время требуется вместо флага `-std=c++20` для компиляции любого кода, использующего модули C++20.
- Флаг компилятора `-x c++-system-header` указывает, что заголовок стандартной библиотеки C++ компилируется в качестве заголовочной единицы. Вы указываете название заголовка без угловых скобок[5].

---

[1] Если эта опция не работает, попробуйте выбрать Preview — Features from the Latest C++ Working Draft (/std:c++latest).

[2] Это может измениться, когда модули C++20 будут полностью реализованы в g++.

[3] «3.23 C++ Modules». Был доступен: 15 сентября 2023 г. https://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Modules.html.

[4] Натан Сидуэлл (Nathan Sidwell), «C++ Modules: A Brief Tour», 19 октября 2020 г. Был доступен: 15 сентября 2023 г. https://accu.org/journals/overload/28/159/sidwell/.

[5] Есть также флаги c++-header и c++-user-header для предварительной компиляции других заголовков в качестве заголовочных блоков. Подробнее: https://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Modules.html.

После такой настройки скомпилируйте файл исходного кода `fig16_01.cpp` следующей командой:

```
g++ -fmodules-ts fig16_01.cpp -o fig16_01
```

Будет создан исполняемый файл с именем `fig16_01`, который вы можете выполнить из командной строки:

```
./fig16_01
```

Компиляция программы с заголовочными единицами в clang++

Используйте следующую команду для компиляции этого примера в clang++ версии 13[1]:

```
clang++ -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps fig16_01.cpp -o fig16_01
```

Флаг `-std=c++20` указывает, что мы используем функции языка C++20.

В некоторых системах стандартная библиотека C++ в версии g++ является библиотекой C++ по умолчанию. Флаг `-stdlib=libc++` гарантирует, что используется стандартная библиотека C++ в версии clang++.

Флаги `-fimplicit-modules` и `-fimplicit-module-maps` позволяют компилятору clang++ генерировать и находить информацию, необходимую для обработки заголовков как заголовочных единиц.

Будет создан исполняемый файл с именем `fig16_01`, который вы можете выполнить из командной строки:

```
./fig16_01
```

16.5. Модули могут уменьшать размеры единиц трансляции и время компиляции

Как уже было сказано, модули могут ускорять компиляцию благодаря исключению повторной предварительной обработки одних и тех же заголовочных файлов во многих единицах трансляции, входящих в программу. Рассмотрим следующую простейшую программу, которая содержит менее 90 символов, включая межстрочные интервалы и отступы:

```
#include <iostream>
int main() {
   std::cout << "Welcome to C++20 Modules!\n";
}
```

---

[1] Это может измениться, когда модули C++20 будут полностью реализованы в clang++. Кроме того, заголовочные единицы работают для стандартных библиотечных заголовков, но пользовательские заголовки требуют создания ваших собственных карт модулей. Дополнительная информация: https://clang.llvm.org/docs/Modules.html#module-maps.

При компиляции этой программы препроцессор вставляет все содержимое заголовка `<iostream>` в единицу трансляции. Каждый из наших компиляторов имеет флаг, позволяющий видеть результат предварительной обработки файла исходного кода:

- `-E` — в g++ и clang++;
- `/P` — в Visual C++.

Мы использовали эти флаги для предварительной обработки предыдущей программы. Размер предварительно обработанных единиц трансляции в нашей системе оказался очень большим:

- 1 023 010 байт в компиляторе g++;
- 1 883 270 байт в компиляторе clang++;
- 1 497 116 байт в компиляторе Visual C++.

Предварительно обработанные единицы трансляции примерно в 11–21 тыс. раз превышают размер исходного файла — это колоссальное увеличение количества байтов на единицу трансляции! Представьте большой проект с тысячами единиц трансляции, каждая из которых включает множество заголовков. Каждая директива `#include` должна быть предварительно обработана для формирования единиц трансляции, которые будут обработаны компилятором.

При использовании заголовочных единиц каждый заголовок обрабатывается только один раз как единица трансляции. Что тоже важно, при обработке заголовочной единицы ее содержимое не вставляется в каждую единицу трансляции. В совокупности это значительно сокращает время компиляции.

Ссылки

Для получения дополнительной информации о размерах единиц трансляции и о быстродействии во время компиляции изучите эти источники:

- Габриэль Дос Рейс (Gabriel Dos Reis) и Кэмерон Да Камара (Cameron DaCamara), «Implementing C++ Modules: Lessons Learned, Lessons Abandoned», 18 декабря 2021 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=9OWGgkuyFV8.
- Бьерн Страуструп (Bjarne Stroustrup), «Thriving in a Crowded and Changing World: C++ 2006–2020 — Section 9.3.1 Modules», 12 июня 2020 г. Был доступен: 15 сентября 2023 г. https://www.stroustrup.com/hopl20main-p5-p-bfc9cd4--final.pdf.
- Кэмерон Да Камара, «Practical C++20 Modules and the Future of Tooling Around C++ Modules», 4 мая 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=ow2zV0Udd9M.
- Райнер Гримм (Rainer Grimm), «C++20: The Advantages of Modules», 10 мая 2020 г. Был доступен: 15 сентября 2023 г. https://www.modernescpp.com/index.php/cpp20-modules.

- Рене Ривера (Rene Rivera), «Are Modules Fast? (Revision 1)», 6 марта 2019 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p1441r1.

Инструменты для профилирования компилятора

Об инструментах, которые можно использовать для измерения быстродействия во время компиляции, вы узнаете из этих источников:

- Кевин Кадье (Kevin Cadieux), Хелена Грегг (Helena Gregg) и Колин Робертсон (Colin Robertson), «Get Started with C++ Build Insights», 3 ноября 2019 г. Был доступен: 15 сентября 2023 г. https://docs.microsoft.com/en-us/cpp/build-insights/get-started-with-cpp-build-insights.
- «Clang 13 Documentation: Target-Independent Compilation Options -ftimereport and -ftime-trace». Был доступен: 15 сентября 2023 г. https://clang.llvm.org/docs/ClangCommandLineReference.html#target-independent-compilation-options.
- «Profiling the C++ Compilation Process». Был доступен: 15 сентября 2023 г. https://stackoverflow.com/questions/13559818/profiling-the-c-compilation-process.

16.6. Пример: создание и использование модуля

Теперь мы определим наш первый модуль. Интерфейс модуля определяет те элементы модуля, которые доступны в других единицах трансляции. Чтобы тот или иной элемент модуля стал доступным в других единицах трансляции, вам надо экспортировать его объявление с помощью ключевого слова `export` одним из четырех способов[1]:

- экспортировать объявление напрямую;
- экспортировать группу объявлений в фигурных скобках ({ и });
- экспортировать пространство имен, которое может содержать множество объявлений;
- экспортировать элемент пространства имен, вместе с которым экспортируется имя этого пространства имен[2], но не другие его элементы.

Напомним, что если первое вхождение идентификатора является его определением, оно также является объявлением идентификатора. Таким образом, каждый элемент в предыдущем списке может экспортировать объявления или определения.

---

[1] Стандарт C++, «10. Modules — 10.2. Export Declaration». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/module.interface.

[2] Таким образом вы квалифицируете экспортированные элементы с помощью «name::».

16.6.1. Объявление модульной единицы

На илл. 16.2 определена наша первая **модульная единица (module unit)**[1] — единица трансляции, которая является частью модуля. Когда модуль состоит из одной единицы трансляции, ее обычно называют просто модулем. В модуль на илл. 16.2 мы включили четыре функции (строки 8–10, 14–16, 21–23 и 28–30), чтобы продемонстрировать **экспортирующие объявления (exporting declarations)**, которые можно использовать в других единицах трансляции. В следующих подразделах мы обсудим экспорт и **«обертки» («wrappers»)** пространства имен для функций в строках 14–16, 21–23 и 28–30.

```
 1 // Fig. 16.2: welcome.ixx
 2 // Основной интерфейс модуля welcome
 3 export module welcome; // Имя модуля
 4
 5 import <string>; // В этом модуле используется класс string
 6
 7 // Экспорт функции
 8 export std::string welcomeStandalone() {
 9    return "welcomeStandalone function called";
10 }
11
12 // Экспорт всех элементов, заключенных в фигурные скобки
13 export {
14    std::string welcomeFromExportBlock() {
15       return "welcomeFromExportBlock function called";
16    }
17 }
18
19 // Экспорт пространства имен экспортирует все элементы в нем
20 export namespace TestNamespace1 {
21    std::string welcomeFromTestNamespace1() {
22       return "welcomeFromTestNamespace1 function called";
23    }
24 }
25
26 // Экспорт элемента также экспортирует имя его пространства имен
27 namespace TestNamespace2 {
28    export std::string welcomeFromTestNamespace2() {
29       return "welcomeFromTestNamespace2 function called";
30    }
31 }
```

Илл. 16.2 | Основной интерфейс модуля welcome

[1] Стандарт C++, «10. Modules — 10.1. Module Units and Purviews». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/module.unit.

Объявления и имена модулей

Объявление модуля в строке 3 присваивает модулю имя `welcome`. Имена модулей — это идентификаторы в нижнем регистре (по соглашению), разделенные точками[1], например `deitel.time` или `deitel.math`, которые мы будем использовать в следующих примерах. Точки не имеют какого-то предназначения, есть даже предложение убрать их из имен модулей[2]. Имена модулей `deitel.time` и `deitel.math` начинаются со слова `deitel`, но эти модули не являются «подмодулями» базового модуля с именем `deitel`. Все объявления, начиная с объявления модуля и до конца единицы трансляции, входят в **область действия модуля (module purview)**, как и объявления из всех других единиц, составляющих модуль[3]. В последующих разделах мы рассмотрим модули, состоящие из нескольких файлов.

Когда вы помещаете перед объявлением модуля ключевое слово `export`, вы объявляете **единицу интерфейса модуля (module interface unit)**, определяющую те элементы, которые будут доступны клиентскому коду. Каждый модуль имеет одну **основную единицу интерфейса (primary module interface unit)**, содержащую объявление `export module` с именем модуля. В разделе 16.9 вы увидите, что основная единица интерфейса модуля может состоять из нескольких раздельных единиц интерфейса **(module interface partition units)**, которые для краткости называют **разделами (partitions)**.

Расширение имен файлов с интерфейсами модулей в Visual C++

В Microsoft Visual C++ принято расширение `.ixx` для имен файлов, содержащих единицы интерфейса модулей. Чтобы добавить такой файл в проект Visual C++:

1. Щелкните правой кнопкой мыши на папку `Source Files` вашего проекта и выберите `Add > Module...`.
2. В диалоговом окне `Add New Item` укажите имя файла (мы использовали `welcome.ixx`) и место для его сохранения, затем нажмите `Add`.
3. Замените код по умолчанию кодом с илл. 16.2.

Вам не обязательно использовать расширение `.ixx`. При другом расширении имени вашего файла, содержащего единицу интерфейса модуля, щелкните правой кнопкой мыши на этот файл в вашем проекте, выберите `Properties` и убедитесь, что в `Item Type` установлено значение `C/C++ compiler`.

Расширения имен файлов с интерфейсами модулей в g++ и clang++

Компиляторы g++ и clang++ не требуют специальных расширений для имен файлов, содержащих единицы интерфейса модулей. В конце этого раздела мы

---

[1] Коррентен Жабо (Corentin Jabot), «Naming Guidelines for Modules», 16 июня 2019 г. Был доступен: 15 сентября 2023 г. https://wg21.link/P1634R0.

[2] Майкл Спенсер (Michael Spencer), «P1873R1: remove.dots.in.module.names», 17 сентября 2019 г. https://wg21.link/p1873r1.

[3] Стандарт C++, «10. Modules — 10.1. Module Units and Purviews».

покажем, как сделать, чтобы g++ и clang++ компилировали файлы `.ixx` и таким образом унифицировать это расширение для работы со всеми тремя компиляторами.

Другие расширения имен файлов

Перечислим распространенные расширения имен файлов, с которыми вы встретитесь при работе с модулями C++20:

- `.ixx` — принятое в Microsoft Visual C++ расширение имен файлов, содержащих основные единицы интерфейса модулей.
- `.ifc` — принятое в Microsoft Visual C++ расширение имен файлов, содержащих скомпилированные версии основных единиц интерфейса модулей[1].
- `.cpp` — расширение имен файлов, содержащих исходный код единиц трансляции, в том числе модульных.
- `.cppm` — рекомендуемое для clang++ расширение имен файлов, содержащих модульные единицы. Visual C++ также распознаёт это расширение.
- `.pcm` — когда вы компилируете основную единицу интерфейса модуля в clang++, результат помещается в файл с этим расширением.

16.6.2. Экспорт объявления

Вы должны экспортировать объявление, чтобы оно было доступно за пределами модуля. В строке 8 на илл. 16.2 к определению функции применяется ключевое слово `export`, которое экспортирует объявление функции (то есть ее прототип) как часть интерфейса модуля. Все экспортируемые объявления должны находиться после объявления модуля в единице трансляции и должны быть доступны либо в области видимости файла (то есть в области видимости глобального пространства имен), либо в именованной области видимости пространства имен (раздел 16.6.5). Объявления в инструкциях `export` не должны иметь **внутреннего связывания (internal linkage)**[2], которое включает:

- статические переменные и функции в области видимости глобального пространства имен в единице трансляции;
- глобальные `const`- или `constexpr`-переменные в единице трансляции;
- идентификаторы, объявленные в «безымянных пространствах имен»[3].

Кроме того, если в модуле определены какие-либо макросы препроцессора, они предназначены только для этого модуля и не могут быть экспортированы[4].

---

[1] Кэмерон Да Камара (Cameron DaCamara), «Practical C++20 Modules and the Future of Tooling Around C++ Modules».

[2] Стандарт C++, «10. Modules — 10.2. Export Declaration».

[3] «Namespaces — Unnamed Namespaces». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/language/namespace#Unnamed_namespaces.

[4] Габриэль Дос Рейс (Gabriel Dos Reis), «Programming with C++ Modules: Guide for the Working Software Developer».

Определения шаблонов, constexpr-функций и inline-функций в модулях

Как и в случае с заголовками, при определении шаблона `constexpr`-функции или `inline`-функции в модуле вы должны экспортировать ее полное определение, чтобы компилятор получил к нему доступ везде, где импортируется модуль.

16.6.3. Экспорт группы объявлений

Вместо экспорта отдельных объявлений вы можете экспортировать группу объявлений, заключенных в фигурные скобки (строки 13–17). Каждое из таких объявлений экспортируется. Фигурные скобки не создают новую область видимости, поэтому все идентификаторы, объявленные в таком блоке, продолжают существовать и после закрывающей фигурной скобки.

16.6.4. Экспорт пространства имен

Программа может содержать множество идентификаторов, определенных в разных областях видимости. Иногда переменная из одной области видимости может «столкнуться» с одноименной переменной из другой области видимости, что приведет к конфликту имен и ошибкам. В C++ эту проблему решают **пространства имен (namespaces)**. Каждое пространство имен определяет свою область видимости, в которой определены идентификаторы и переменные. Как вы знаете, функции стандартной библиотеки C++ определены в пространстве имен `std`. Это гарантирует, что идентификаторы этой библиотеки не конфликтуют с идентификаторами в вашем собственном коде.

Определение и экспорт пространств имен

В строках 20–24 определяется пространство имен `TestNamespace1`. Тело пространства имен ограничено фигурными скобками (`{}`). Пространство имен может содержать константы, данные, классы и функции. Определения пространств имен должны быть размещены в области видимости глобального пространства имен или вложены в другие пространства имен. В отличие от классов, элементы данных пространства имен могут быть определены в отдельных, но идентично именованных блоках пространства имен. Например, каждый заголовок стандартной библиотеки C++ содержит блок пространства имен такого вида:

```
namespace std {
   // Объявления заголовка стандартной библиотеки
}
```

Он указывает, что объявления заголовка находятся в пространстве имен `std`. Когда вы ставите ключевое слово `export` перед блоком пространства имен, экспортируются все элементы этого блока, но не экспортируются элементы других блоков этого же пространства имен.

Доступ к элементам пространства имен

Как обычно, для использования элемента пространства имен вы должны квалифицировать этот элемент, добавив имя пространства имен и оператор разрешения области видимости (`::`), например:

```
TestNamespace1::welcomeFromTestNamespace1()
```

В качестве альтернативы вы можете использовать объявление `using` или директиву `using` до того, как элемент будет использован в единице трансляции. Пример объявления `using`:

```
using TestNamespace1::welcomeFromTestNamespace1;
```

Это объявление добавляет один идентификатор (`welcomeFromTestNamespace1`) в ту область видимости, в которой оно находится. Пример директивы `using`:

```
using namespace TestNamespace1;
```

Эта директива добавляет все идентификаторы из указанного пространства имен в ту область видимости, в которой она находится. Элементы одного и того же пространства имен могут обращаться друг к другу напрямую, не используя квалификатор пространства имен.

16.6.5. Экспорт элемента пространства имен

Вы можете экспортировать не все пространство имен целиком, а только некоторые его элементы, как показано в строках 27–31. В этом случае имя пространства имен тоже экспортируется, но это не приводит к неявному экспорту других элементов пространства имен.

16.6.6. Импорт модуля для его использования в экспорте объявлений

Чтобы использовать экспортированные объявления модуля в вашей единице трансляции, вы должны предоставить объявление `import` с именем этого модуля (илл. 16.3, строка 4) в области видимости глобального пространства имен. Экспортированные объявления модуля будут доступны от объявления `import` до конца единицы трансляции. Импорт модуля не добавляет его код в единицу трансляции. В отличие от заголовков, модулям не требуется предотвращение множественных включений (раздел 9.7.3). В строках 7–10 вызываются четыре функции, экспортированные из модуля `welcome` (илл. 16.2). Для функций, определенных в пространствах имен, в строках 9 и 10 перед именем каждой функции указано ее пространство имен и оператор разрешения области видимости (`::`). Выходные данные программы показывают, что мы смогли вызвать все функции, которые экспортирует модуль `welcome`.

```
// fig16_03.cpp
// Импорт модуля и использование его экспортированных элементов
import <iostream>;
import welcome; // Импортируем модуль welcome

int main() {
   std::cout << welcomeStandalone() << '\n'
      << welcomeFromExportBlock() << '\n'
      << TestNamespace1::welcomeFromTestNamespace1() << '\n'
      << TestNamespace2::welcomeFromTestNamespace2() << '\n';
}
```

```
welcomeStandalone function called
welcomeFromExportBlock function called
welcomeFromTestNamespace1 function called
welcomeFromTestNamespace2 function called
```

Илл. 16.3 | Импорт модуля и использование его экспортированных элементов

Компиляция примера в Visual C++

Убедитесь, что файл `fig16_03.cpp` находится в папке `Source Files` вашего проекта, и запустите проект, чтобы скомпилировать модуль и программу с функцией `main`.

Компиляция примера в g++

Выполните следующие команды (они могут измениться, когда в компиляторе g++ будет реализована полная поддержка модулей C++20):

1. Скомпилируйте заголовки `<string>` и `<iostream>` как заголовочные единицы, поскольку они используются в нашем модуле и в нашей программе с функцией `main`:

   ```
   g++ -fmodules-ts -x c++-system-header string
   g++ -fmodules-ts -x c++-system-header iostream
   ```

2. Скомпилируйте единицу интерфейса модуля, чтобы получить файл `welcome.o`:

   ```
   g++ -fmodules-ts -c -x c++ welcome.ixx
   ```

3. Скомпилируйте программу с функцией `main` и **скомпонуйте (link)** ее с файлом `welcome.o`, чтобы получить исполняемый файл `fig16_03`:

   ```
   g++ -fmodules-ts fig16_03.cpp welcome.o -o fig16_03
   ```

На шаге 2:

- параметр `-c` означает, что надо скомпилировать файл `welcome.ixx`, но не компоновать его с другими файлами;
- параметр `-x c++` означает, что `welcome.ixx` является файлом C++.

Параметр `-x c++` необходим, так как расширение `.ixx` не является стандартным для компилятора g++. Если переименовать файл `welcome.ixx` в `welcome.cpp`, то параметр `-x c++` не потребуется.

Компиляция примера в clang++

Выполните следующие команды (они могут измениться, когда в компиляторе clang++ будет реализована полная поддержка модулей C++20):

1. Скомпилируйте единицу интерфейса модуля в файл `.pcm` («precompiled module» — формат, специфичный для clang++):

```
clang++ -c -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -x c++ welcome.ixx
   -Xclang -emit-module-interface -o welcome.pcm
```

2. Скомпилируйте программу с функцией `main` и скомпонуйте ее с файлом `welcome.pcm`, чтобы получить исполняемый файл `fig16_03`:

```
clang++ -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -fprebuilt-module-path=.
   fig16_03.cpp welcome.pcm -o fig16_03
```

На шаге 1:

- параметр `-c` означает, что надо скомпилировать файл `welcome.ixx`, но не компоновать его с другими файлами;
- параметр `-x c++` означает, что `welcome.ixx` является файлом C++.

Параметр `-x c++` необходим, так как расширение `.ixx` не является стандартным для компилятора clang++. Если переименовать файл `welcome.ixx` в `welcome.cpp`, то параметр `-x c++` не потребуется. На шаге 2 параметр:

```
-fprebuilt-module-path=.
```

указывает расположение скомпилированного файла модуля (`.pcm`). Точка (`.`) — это текущая папка, но ее можно заменить относительным или полным путем к другой папке в вашей системе.

16.6.7. Пример: попытка доступа к неэкспортированному элементу модуля

Модули не экспортируют объявления неявно, это называется строгой инкапсуляцией. Таким образом, вы получаете полный контроль над экспортом всех объявлений, используемых в единицах трансляции. На илл. 16.4 определена основная единица интерфейса модуля (строка 3) с именем `deitel.math`, содержащая пространство имен `deitel::math`[1], которое не экспортируется. В этом пространстве имен мы определяем две функции: `square` (строки 7–9), которая экспортируется, и `cube` (строки 12–14), которая не экспортируется. Экспорт функции `square` неявно экспортирует имя пространства имен, содержащего эту функцию, но не экспортирует другие элементы этого пространства имен. Поскольку функция `cube` не экспортируется, другие единицы трансляции не могут ее вызывать (как будет показано на илл. 16.5). Это

[1] Пространство имен deitel::math определяет пространство имен deitel с вложенным пространством имен math. Используемый здесь синтаксис был введен в C++17.

важнейшее отличие от заголовков! Все, что объявлено в заголовке, может быть использовано везде, где вы его включаете директивой `#include`[1].

```
1 // Fig. 16.4: deitel.math.ixx
2 // Основной интерфейс модуля deitel.math.
3 export module deitel.math; // Имя модуля
4
5 namespace deitel::math {
6    // Экспорт функции square и всего пространства имен deitel::math (неявно)
7    export int square(int x) {
8       return x * x;
9    }
10
11   // Неэкспортированная функция cube не экспортируется неявно
12   int cube(int x) {
13      return x * x * x;
14   }
15 };
```

Илл. 16.4 | Основной интерфейс модуля deitel.math

Размещение экспортируемых идентификаторов в пространствах имен — это хорошая практика, позволяющая избежать коллизии имен, когда несколько модулей экспортируют идентификаторы с одинаковыми именами. Наше исследование показало, что программисты обычно называют пространства имен подобно именам их модулей[2], поэтому и в нашем модуле `deitel.math` пространство имен называется `deitel::math` (строка 5).

В коде на илл. 16.5 в строке 4 импортируется модуль `deitel.math`. В строке 9 вызывается функция `deitel::math::square`, имя которой квалифицировано тем пространством имен, где она находится. Эта строка компилируется, потому что модуль `deitel.math` экспортирует функцию `square`. Но в строке 12 возникают ошибки компиляции, потому что модуль `deitel.math` не экспортирует функцию `cube`. В выходных данных показаны сообщения компилятора Visual C++. Мы выделили ключевые сообщения об ошибках жирным шрифтом и добавили пустые строки для удобства чтения.

```
1 // fig16_05.cpp
2 // Демонстрация недоступности неэкспортированных идентификаторов модуля
3 import <iostream>;
4 import deitel.math; // Импортируем модуль deitel.math
5
6 int main() {
```

[1] Габриэль Дос Рейс (Gabriel Dos Reis), «Programming with C++ Modules: Guide for the Working Software Developer».

[2] Даниэла Энгерт (Daniela Engert), «Modules: The Beginner's Guide», 2 мая 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=Kqo-jIq4V3I.

```
    // Вызов функции square возможен, так как она экспортирована из пространства
    // имен deitel::math, что неявно экспортирует все это пространство имен
    std::cout << "square(3) = " << deitel::math::square(3) << '\n';

    // Вызов функции cube невозможен, так как она не экспортирована
    std::cout << "cube(3) = " << deitel::math::cube(3) << '\n';
}
```

```
Build started...
1>------ Build started: Project: modules_demo, Configuration: Debug Win32 ---
---
1>Scanning sources for module dependencies...
1>deitel.math.ixx
1>fig16_05.cpp

1>Compiling...
1>deitel.math.ixx
1>fig16_05.cpp

1>C:\Users\pauldeitel\Documents\examples\ch16\fig16_04-
05\fig16_05.cpp(12,47): error C2039: 'cube': is not a member of 'deitel::math'

1>C:\Users\pauldeitel\Documents\examples\examples\ch16\fig16_04-05\deitel.
math.ixx(5): message : see declaration of 'deitel::math'

1>C:\Users\pauldeitel\Documents\examples\examples\ch16\fig16_04-
05\fig16_05.cpp(12,51): error C3861: 'cube': identifier not found
```

Илл. 16.5 | Демонстрация недоступности неэкспортированных идентификаторов модуля

Сообщения компилятора g++ об ошибках

Чтобы увидеть сообщения об ошибках, выполните следующие команды, каждая из которых описана в разделе 16.6.6:

1. `g++ -fmodules-ts -x c++-system-header iostream`
2. `g++ -fmodules-ts -c -x c++ deitel.math.ixx`
3. `g++ -fmodules-ts fig16_05.cpp deitel.math.o`

Мы выделили жирным шрифтом ключевое сообщение об ошибке:

```
fig16_05.cpp: In function 'int main()':
fig16_05.cpp:12:49: error: 'cube' is not a member of 'deitel::math'
   12 | std::cout << "cube(e) = " << deitel::math::cube(3) << '\n';
      |                                                 ^~~~
```

Сообщения компилятора clang++ об ошибках

Чтобы увидеть сообщения об ошибках, выполните следующие команды, каждая из которых описана в разделе 16.6.6:

1. `clang++ -c -std=c++20 -stdlib=libc++ -fimplicit-modules -fimplicit-module-maps -x c++ deitel.math.ixx -Xclang -emit-module-interface -o deitel.math.pcm`
2. `clang++ -std=c++20 -stdlib=libc++ -fimplicit-modules -fimplicit-module-maps -fprebuilt-module-path=. fig16_05.cpp deitel.math.pcm -o fig16_05`

Мы выделили жирным шрифтом ключевое сообщение об ошибке:

```
fig16_05.cpp:12:49: error: no member named 'cube' in namespace 'deitel::math'
   std::cout << "cube(e) = " << deitel::math::cube(3) << '\n';
                               ~~~~~~~~~~~~~~~^
1 error generated.
```

16.7. Глобальный фрагмент модуля

Напомним, что некоторые заголовки нельзя скомпилировать как заголовочные единицы, поскольку они зависят от состояния препроцессора, например, содержат макросы, определенные в вашей единице трансляции, или другие заголовки. Такие заголовки можно включить директивой `#include` в **глобальный фрагмент модуля (global module fragment)**[1], который находится в самом начале модуля, перед его объявлением, и начинается со следующей строки:

```
module;
```

Глобальный фрагмент модуля может содержать только директивы препроцессора. Единица интерфейса модуля может экспортировать объявления, включенные директивой `#include` в глобальный фрагмент модуля, поэтому их могут использовать другие единицы, импортирующие этот модуль[2]. Глобальные фрагменты из всех модульных единиц помещаются в **глобальный модуль (global module)**, который также содержит не относящийся к модулям код немодульных единиц трансляции, таких как файл с функцией `main`.

16.8. Отделение интерфейса от реализации

В главах 9 и 10 мы определяли классы с помощью заголовочных файлов с расширением `.h` и файлов исходного кода с расширением `.cpp`, чтобы отделять интерфейс класса от его реализации. Модули тоже поддерживают отделение интерфейса от реализации:

---

[1] Стандарт C++, «10. Modules — 10.4. Global Module Fragment». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/module.global.frag.

[2] Габриэль Дос Рейс (Gabriel Dos Reis), «Programming with C++ Modules: Guide for the Working Software Developer».

- Вы можете отделить интерфейс от реализации, помещая их в разные файлы (разделы 16.8.1–16.8.2).
- Вы можете определить интерфейс и реализацию в одном файле (раздел 16.8.3).

16.8.1. Пример: единицы реализации модуля

Иногда полезно разделять большой модуль на маленькие и хорошо управляемые части: например, когда команда разработчиков работает над разными фрагментами одного и того же модуля. Определение модуля можно разделить на несколько файлов с исходным кодом. Давайте поместим весь интерфейс модуля в основную единицу интерфейса модуля, а все детали реализации модуля — в отдельный файл.

Основная единица интерфейса модуля

Основная единица интерфейса модуля `deitel.math` (илл. 16.6) экспортирует пространство имен `deitel::math` (строки 7–10), содержащее прототип функции `average`, вычисляющей среднее значение элементов `vector<int>`.

```
 1 // Fig. 16.6: deitel.math.ixx
 2 // Основной интерфейс модуля deitel.math
 3 export module deitel.math; // Имя модуля
 4
 5 import <vector>;
 6
 7 export namespace deitel::math {
 8    // Вычисляем среднее арифметическое элементов vector<int>
 9    double average(const std::vector<int>& values);
10 };
```

Илл. 16.6 | Основной интерфейс модуля deitel.math

Единица реализации модуля

Все файлы, содержащие объявления модулей без ключевого слова `export` (как в строке 3 на илл. 16.7), являются единицами реализации модуля (обычно определенными в файлах с расширением `.cpp`)[1]. Большой файл реализации можно разделить на несколько файлов, чтобы упростить работу над кодом. В строке 3 указано, что этот файл является единицей реализации модуля `deitel.math`. Единица реализации модуля неявно импортирует интерфейс модуля с указанным именем. Компилятор объединяет основную единицу интерфейса модуля и соответствующую единицу реализации модуля (одну или несколько) в единый именованный модуль, который могут импортировать другие единицы трансля-

[1] Стандарт C++20, «10 Modules – 10.1 Module Units and Purviews». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/module#unit.

ции[1]. В строках 10–13 функция average реализована в том же пространстве имен, где находится ее прототип в основной единице интерфейса модуля (илл. 16.6).

```
1 // Fig. 16.7: deitel.math-impl.cpp
2 // Единица реализации модуля deitel.math
3 module deitel.math; // Содержимое этого файла относится к модулю deitel.math
4
5 import <numeric>;
6 import <vector>;
7
8 namespace deitel::math {
9    // Реализация функции average
10   double average(const std::vector<int>& values) {
11      double total{std::accumulate(values.begin(), values.end(), 0.0)};
12      return total / values.size();
13   }
14 };
```

Илл. 16.7 | Единица реализации модуля deitel.math

Работа с модулем

Программа с функцией main на илл. 16.8 импортирует модуль deitel.math (строка 7) и вызывает его функцию average (строка 17) для вычисления среднего значения элементов вектора integers.

```
1 // fig16_08.cpp
2 // Использование функции average модуля deitel.math
3 import <algorithm>;
4 import <iostream>;
5 import <iterator>;
6 import <vector>;
7 import deitel.math; // Импортируем модуль deitel.math
8
9 int main() {
10   std::ostream_iterator<int> output(std::cout, " ");
11   std::vector integers{1, 2, 3, 4};
12
13   std::cout << "vector integers: ";
14   std::copy(integers.begin(), integers.end(), output);
15
16   std::cout << "\naverage of integer's elements: "
17      << deitel::math::average(integers) << '\n';
18 }
```

```
vector integers: 1 2 3 4
average of integer's elements: 2.5
```

Илл. 16.8 | Использование функции average модуля deitel.math

[1] Стандарт C++20, «10 Modules». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/module.

Компиляция примера в Visual C++

Добавьте файл `deitel.math.ixx` в проект Visual C++, как показано в разделе 16.6.1. Убедитесь, что в папке `Source Files` вашего проекта находятся файлы:

- `deitel.math.ixx` — основная единица интерфейса модуля;
- `deitel.math-impl.cpp`[1] — единица реализации модуля;
- `fig16_08.cpp` — программа с функцией `main`.

Затем запустите проект, чтобы скомпилировать модуль и программу с функцией `main`.

Компиляция примера в g++

Выполните команды, приведенные в разделе 16.6.6, для компиляции заголовков стандартной библиотеки `<algorithm>`, `<iostream>`, `<iterator>`, `<numeric>` и `<vector>` в качестве заголовочных единиц. Затем скомпилируйте основную единицу интерфейса модуля:

```
g++ -fmodules-ts -c -x c++ deitel.math.ixx
```

Скомпилируйте единицу реализации модуля:

```
g++ -fmodules-ts -c deitel.math-impl.cpp
```

Вы получили объектные файлы `deitel.math.o` и `deitel.math-impl.o`, содержащие интерфейс и реализацию модуля соответственно. Скомпилируйте программу с функцией `main` и скомпонуйте ее с файлами `deitel.math.o` и `deitel.math-impl.o`:

```
g++ -fmodules-ts fig16_08.cpp deitel.math.o deitel.math-impl.o
   -o fig16_08
```

Компиляция примера в clang++

Скомпилируйте основную единицу интерфейса модуля в файл `.pcm`:

```
clang++ -c -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -x c++ deitel.math.ixx
   -Xclang -emit-module-interface -o deitel.math.pcm
```

Скомпилируйте единицу реализации модуля в объектный файл:

```
clang++ -c -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -fmodule-file=deitel.math.pcm
   deitel.math-impl.cpp
```

Параметр `-fmodule-file=deitel.math.pcm` задает имя основной единицы интерфейса модуля. Скомпилируйте программу с функцией `main` и скомпонуйте ее с файлами `deitel.math-impl.o` и `deitel.math.pcm`:

---

[1] Для совместимости с этапами компиляции g++ мы добавили «-impl» к имени файла с реализацией модуля, чтобы у каждого файла с расширением .o было уникальное имя.

```
clang++ -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -fprebuilt-module-path=. fig16_08.cpp
   deitel.math-impl.o deitel.math.pcm -o fig16_08
```

16.8.2. Пример: класс на основе модулей

Давайте определим упрощенную версию класса `Time` из главы 9, поместив его интерфейс в основную единицу интерфейса модуля, а реализацию — в единицу реализации модуля. Мы назовем этот модуль `deitel.time` и поместим класс в пространство имен `deitel::time`.

Основная единица интерфейса модуля deitel.time

Основная единица интерфейса модуля `deitel.time` (илл. 16.9) определяет и экспортирует пространство имен `deitel::time` (строки 7–19), содержащее определение класса `Time` (строки 8–18).

```
1 // Fig. 16.9: deitel.time.ixx
2 // Основной интерфейс модуля для упрощенного класса Time
3 export module deitel.time; // Объявляем основной интерфейс модуля
4
5 import <string>; // Вместо #include <string>
6
7 export namespace deitel::time {
8    class Time {
9    public:
10       // Конструктор по умолчанию, так как он может быть вызван без аргументов
11       explicit Time(int hour = 0, int minute = 0, int second = 0);
12
13       std::string toString() const;
14    private:
15       int m_hour{0}; // 0 - 23 (24-часовой формат)
16       int m_minute{0}; // 0 - 59
17       int m_second{0}; // 0 - 59
18    };
19 }
```

Илл. 16.9 | Основной интерфейс модуля для упрощенного класса Time

Единица реализации модуля deitel.time

Объявление `module` в строке 4 на илл. 16.10 указывает, что файл `deitel.time-impl.cpp` содержит реализацию модуля `deitel.time`. В остальной части этого файла определены функции класса `Time`. Директива `using` в строке 8 предоставляет единице реализации модуля доступ к пространству имен `deitel::time`:

```
using namespace deitel::time;
```

Однако любая единица трансляции, импортирующая модуль `deitel.time`, не видит эту директиву `using`. Таким образом, другие единицы трансляции по-прежнему

должны получать доступ к экспортированным именам нашего модуля через `deitel::time` или через собственные директивы `using`.

```
// Fig. 16.10: deitel.time-impl.cpp
// Единица реализации модуля deitel.time, содержащая
// определения функций класса Time
module deitel.time; // Единица реализации модуля deitel.time

import <stdexcept>;
import <string>;
using namespace deitel::time;

// Конструктор класса Time инициализирует каждый элемент данных
Time::Time(int hour, int minute, int second) {
   // Проверяем аргументы hour, minute и second
   if ((hour < 0 || hour >= 24) || (minute < 0 || minute >= 60) ||
      (second < 0 || second >= 60)) {
      throw std::invalid_argument{
         "hour, minute or second was out of range"};
   }

   m_hour = hour;
   m_minute = minute;
   m_second = second;
}

// Возвращаем строковое представление объекта Time
std::string Time::toString() const {
   using namespace std::string_literals;

   return "Hour: "s + std::to_string(m_hour) +
      "\nMinute: "s + std::to_string(m_minute) +
      "\nSecond: "s + std::to_string(m_second);
}
```

Илл. 16.10 | Единица реализации модуля deitel.time, содержащая определения функций класса Time

Работа с классом Time из модуля deitel.time

Программа, показанная на илл. 16.11, импортирует модуль `deitel.time` (строка 7) и использует класс `Time`. Для удобства в строке 8 указано, что эта программа использует пространство имен модуля `deitel::time`, но вы можете полностью квалифицировать имя класса `Time` в строках 11 и 17: `deitel::time::Time`.

```
// fig16_11.cpp
// Импорт модуля deitel.time и использование его класса Time
import <iostream>;
import <stdexcept>;
import <string>;
```

```
 6
 7 import deitel.time;
 8 using namespace deitel::time;
 9
10 int main() {
11    const Time t{12, 25, 42}; // Указываем hour, minute и second
12
13    std::cout << "Time t:\n" << t.toString() << "\n\n";
14
15    // Пробуем инициализировать t2 недопустимыми значениями
16    try {
17       const Time t2{27, 74, 99}; // Все три значения ошибочны!
18    }
19    catch (const std::invalid_argument& e) {
20       std::cout << "t2 not created: " << e.what() << '\n';
21    }
22 }
```

```
Time t:
Hour: 12
Minute: 25
Second: 42

t2 not created: hour, minute or second was out of range
```

Илл. 16.11 | Импорт модуля deitel.time и использование его класса Time

Компиляция примера в Visual C++

Добавьте файл `deitel.time.ixx` в проект Visual C++, как показано в разделе 16.6.1. Убедитесь, что в папке `Source Files` вашего проекта находятся файлы:

- `deitel.time.ixx` — основная единица интерфейса модуля;
- `deitel.time-impl.cpp` — единица реализации модуля;
- `fig16_11.cpp` — программа с функцией `main`.

Затем запустите проект, чтобы скомпилировать модуль и программу с функцией `main`.

Компиляция примера в g++

Выполните команды, приведенные в разделе 16.6.6, для компиляции заголовков стандартной библиотеки `<iostream>`, `<string>` и `<stdexcept>` в качестве заголовочных единиц. Затем скомпилируйте основную единицу интерфейса модуля:

```
g++ -fmodules-ts -c -x c++ deitel.time.ixx
```

Скомпилируйте единицу реализации модуля:

```
g++ -fmodules-ts -c deitel.time-impl.cpp
```

Вы получили файлы `deitel.math.o` и `deitel.math-impl.o`, содержащие интерфейс и реализацию модуля соответственно. Скомпилируйте программу с функцией `main` и скомпонуйте ее с файлами `deitel.time.o` и `deitel.time-impl.o`[1]:

```
g++ -fmodules-ts fig16_11.cpp deitel.time.o deitel.time-impl.o
   -o fig16_11
```

Компиляция примера в clang++

Скомпилируйте основную единицу интерфейса модуля в файл `.pcm`:

```
clang++ -c -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -x c++ deitel.time.ixx
   -Xclang -emit-module-interface -o deitel.time.pcm
```

Скомпилируйте единицу реализации модуля в объектный файл:

```
clang++ -c -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -fmodule-file=deitel.time.pcm
   deitel.time-impl.cpp
```

Параметр `-fmodule-file=deitel.time.pcm` задает имя основной единицы интерфейса модуля. Скомпилируйте программу с функцией `main` и скомпонуйте ее с файлами `deitel.time-impl.o` и `deitel.time.pcm`:

```
clang++ -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -fprebuilt-module-path=.
   fig16_11.cpp deitel.time-impl.o deitel.time.pcm -o fig16_11
```

16.8.3. Фрагмент :private

Модули поддерживают отделение интерфейса от реализации в одной единице трансляции. Для этого предназначен фрагмент `:private` в основной единице интерфейса модуля[2,3,4]:

```
export module name; // Имя модуля
// Код, определяющий основной интерфейс модуля
// фрагмент :private, определяющий детали реализации
module :private;
// Детали реализации
```

---

[1] На момент написания книги этот пример компоновался в g++ неправильно.

[2] Стандарт C++20, «10 Modules — Private module fragment». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/module.private.frag.

[3] Кэмерон Да Камара (Cameron DaCamara), «Standard C++20 Modules Support with MSVC in Visual Studio 2019 Version 16.8 — Private Module Fragments», 14 сентября 2020 г. Был доступен: 15 сентября 2023 г. https://devblogs.microsoft.com/cppblog/standard-c20-modules-support-with-msvc-in-visualstudio-2019-version-16-8/#private-module-fragments.

[4] Габриэль Дос Рейс (Gabriel Dos Reis), «Programming with C++ Modules: Guide for the Working Software Developer».

При таком подходе основная единица интерфейса модуля должна быть единственной единицей модуля. Изменения в деталях реализации, размещенных внутри фрагмента `:private`, не влияют на интерфейс модуля и на другие единицы трансляции, импортирующие этот модуль[1].

Кэмерон Да Камара из команды Microsoft Visual C++ рекомендует: «Используйте фрагмент модуля `:private`, если вы хотите объединить весь ваш скомпилированный код вместе с кодом интерфейса в одной единице трансляции. По сути, я рассматриваю фрагмент модуля `:private` как единицу реализации модуля, определенную после `module :private;`, но мне не приходится компилировать отдельный файл `.cpp`, чтобы реализовать детали интерфейса. Есть и другое важное преимущество: если весь код находится в одном интерфейсе, это может помочь вам принимать решения об оптимизации вашего инструментария (возможно, улучшая его), не прибегая к запутанной технологии с компоновщиком»[2].

16.9. Разделы модуля

Вы можете разделить интерфейс модуля и (или) его реализацию на несколько частей, называемых **разделами (partitions)**[3] или **раздельными единицами (partition units)**. При работе над крупными проектами это может помочь организовать компоненты модуля в маленькие, хорошо управляемые единицы трансляции. Разделение крупного модуля на меньшие единицы трансляции также может ускорить компиляцию в больших системах. Только те единицы трансляции, которые были изменены, и единицы трансляции, зависящие от этих изменений, должны быть перекомпилированы[4]. Какие элементы требуют перекомпиляции, зависит от имеющихся в компиляторе средств поддержки модулей[5]. Компилятор объединяет разделы модуля в один именованный модуль для импорта в другие единицы трансляции.

16.9.1. Пример: разделы интерфейса модуля

В этом примере мы создадим модуль `deitel.math`, экспортирующий в свою основную единицу интерфейса четыре функции: `square`, `cube`, `squareRoot` и `CubeRoot`. Мы разделим эти функции на два раздела (`powers` и `roots`), чтобы показать синтаксис разделов. Затем мы объединим их экспортированные объявления в единый раздел основной единицы интерфейса модуля.

---

[1] Стандарт C++20, «10 Modules – Private Module Fragment». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/module.private.frag.

[2] Цитата из переписки, состоявшейся 10 февраля 2022 г. между Полом Дейтелом и Кэмероном Да Камара, старшим инженером-программистом Microsoft.

[3] На момент написания книги компилятор clang++ еще не поддерживал разделы.

[4] Кэмерон Да Камара (Cameron DaCamara), «Practical C++20 Modules and the Future of Tooling Around C++ Modules».

[5] Средства поддержки модулей в компиляторах на момент написания книги находятся в стадии разработки и будут развиваться в течение нескольких лет.

Раздел интерфейса модуля deitel.math:powers

В строке 3 на илл. 16.12 указано, что единица трансляции `deitel.math-powers.ixx` является разделом интерфейса модуля. Обозначение `deitel.math:powers` указывает, что раздел называется `powers` и является частью модуля `deitel.math`. Этот раздел экспортирует пространство имен `deitel::math`, содержащее функции `square` и `cube`. Разделы модуля не видны за его пределами, поэтому их нельзя импортировать в единицы трансляции, не входящие в этот модуль[1].

```
1 // Fig. 16.12: deitel.math-powers.ixx
2 // Раздел интерфейса модуля deitel.math:powers
3 export module deitel.math:powers;
4
5 export namespace deitel::math {
6    double square(double x) {return x * x;}
7    double cube(double x) {return x * x * x;}
8 }
```

Илл. 16.12 | Раздел интерфейса модуля deitel.math:powers

Раздел интерфейса модуля deitel.math:roots

В строке 3 на илл. 16.13 указано, что единица трансляции `deitel.math-roots.ixx` является разделом интерфейса модуля. Имя этого раздела — `roots`, он является частью модуля `deitel.math`. Этот раздел экспортирует пространство имен `deitel::math`, содержащее функции `squareRoot` и `CubeRoot`. Есть несколько правил работы с разделами:

- Все разделы интерфейса модуля с одним и тем же именем модуля (в данном случае `deitel.math`) являются его частями. Они не известны друг другу неявным образом, и они не импортируют неявным образом интерфейс модуля[2].
- Разделы можно импортировать только в другие единицы трансляции этого же модуля.
- Раздел интерфейса модуля может импортировать другой раздел интерфейса этого же модуля, чтобы использовать его функции.

```
1 // Fig. 16.13: deitel.math-roots.ixx
2 // Раздел интерфейса модуля deitel.math:roots
3 export module deitel.math:roots;
4
5 import <cmath>;
6
```

Окончание ⇨

[1] «Modules — Module Partitions». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/language/modules.

[2] Габриэль Дос Рейс (Gabriel Dos Reis), «Programming with C++ Modules: Guide for the Working Software Developer».

```
 7 export namespace deitel::math {
 8    double squareRoot(double x) { return std::sqrt(x); }
 9    double cubeRoot(double x) { return std::cbrt(x); }
10 }
```

Илл. 16.13 | Раздел интерфейса модуля deitel.math:roots

Основной интерфейс модуля deitel.math

На илл. 16.14 определена основная единица интерфейса модуля `deitel.math.ixx`. Каждый модуль должен иметь одну основную единицу интерфейса с объявлением `export module`, которое не включает имя раздела (строка 4). В строках 8 и 9 импортируются и экспортируются разделы интерфейса модуля:

- За каждым ключевым словом `import` следует двоеточие (`:`) и имя раздела интерфейса (в данном случае `powers` или `roots`).
- Размещение ключевого слова `export` перед ключевым словом `import` означает, что экспортируемые элементы каждого раздела интерфейса также должны быть частями основного интерфейса модуля `deitel.math`.

Пользователи модуля не имеют доступа к его разделам[1].

```
 1 // Fig. 16.14: deitel.math.ixx
 2 // Основной интерфейс модуля deitel.math, экспортирующий объявления
 3 // из разделов интерфейса :powers и :roots
 4 export module deitel.math; // Объявляем основной интерфейс модуля
 5
 6 // Импортируем и реэкспортируем объявления из
 7 // разделов интерфейса :powers и :roots
 8 export import :powers;
 9 export import :roots;
```

Илл. 16.14 | Основной интерфейс модуля deitel.math, экспортирующий объявления из разделов интерфейса :powers и :roots

Вы также можете экспортировать импортированную основную единицу интерфейса модуля. Предположим, что у нас есть модули с именами `A` и `B`, причем основная единица интерфейса модуля `A` содержит инструкцию:

```
export import B;
```

В этом случае единица трансляции, импортирующая модуль `A`, также импортирует модуль `B` и может использовать его экспортированные объявления.

Если вы экспортируете импортированную заголовочную единицу, ее макросы препроцессора будут доступны для использования только в импортирующей единице трансляции — они не экспортируются повторно. Поэтому для использования

[1] Стандарт C++, «10. Modules — 10.1. Module Units and Purviews». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/module.unit.

макроса из заголовочной единицы в какой-либо другой единице трансляции вы должны явно импортировать заголовок.

Работа с модулем deitel.math

Программа, показанная на илл. 16.15, импортирует модуль `deitel.math` (строка 4). В строках 9–12 используются его экспортированные функции, поэтому мы видим, что основной интерфейс модуля содержит все функции из разделов интерфейса `powers` и `roots`.

```
1 // fig16_15.cpp
2 // Использование функций модуля deitel.math
3 import <iostream>;
4 import deitel.math; // Импортируем модуль deitel.math
5
6 using namespace deitel::math;
7
8 int main() {
9    std::cout << "square(6): " << square(6)
10       << "\ncube(5): " << cube(5)
11       << "\nsquareRoot(9): " << squareRoot(9)
12       << "\ncubeRoot(1000): " << cubeRoot(1000) << '\n';
13 }
```

```
square(6): 36
cube(5): 125
squareRoot(9): 3
cubeRoot(1000): 10
```

Илл. 16.15 | Использование функций модуля deitel.math

Компиляция примера в Visual C++

Добавьте файлы `deitel.math-powers.ixx`, `deitel.math-roots.ixx` и `deitel.math.ixx` в проект Visual C++, как показано в разделе 16.6.1. Добавьте файл `fig16_15.cpp` в папку `Source Files` вашего проекта. Запустите проект, чтобы скомпилировать модуль и программу с функцией `main`.

Компиляция примера в g++

При создании модуля с разделами вы должны сначала скомпилировать разделы, а только потом — основную единицу интерфейса модуля. Выполните команды, приведенные в разделе 16.6.6, для компиляции заголовков стандартной библиотеки `<cmath>` и `<iostream>` в качестве заголовочных единиц. Затем скомпилируйте каждый из разделов интерфейса:

```
g++ -fmodules-ts -c -x c++ deitel.math-powers.ixx
g++ -fmodules-ts -c -x c++ deitel.math-roots.ixx
```

Скомпилируйте основную единицу интерфейса модуля:

```
g++ -fmodules-ts -c -x c++ deitel.math.ixx
```

Скомпилируйте программу с функцией `main` и скомпонуйте ее с файлами `deitel.math-powers.o`, `deitel.math-roots.o` и `deitel.math.o`:

```
g++ -fmodules-ts fig16_15.cpp deitel.math-powers.o
   deitel.math-roots.o deitel.math.o -o fig16_15
```

16.9.2. Разделы реализации модуля

Код реализации модуля тоже можно разделить на несколько файлов, которые будут называться разделами или раздельными единицами реализации[1]. Опять же, это может помочь организовать компоненты модуля в маленькие, хорошо управляемые единицы трансляции, а также ускорить компиляцию в больших системах. В разделе реализации объявление `module` не должно содержать ключевого слова `export`:

```
module ИмяМодуля:ИмяРаздела;
```

Разделы реализации модуля не импортируют неявным образом основной интерфейс модуля[2]. На момент написания книги ни один из трех наших компиляторов еще не поддерживал разделы реализации модуля, поэтому мы их не рассматриваем.

16.9.3. Пример: «подмодули» и разделы

Некоторые библиотеки довольно большие, например стандартная библиотека C++. Программистам, работающим с такой библиотекой, может быть нужно импортировать только какую-то ее часть. Поставщик библиотеки может разделить ее на логические «подмодули» («submodules»), каждый из которых независимо импортируется и имеет собственную основную единицу интерфейса. Кроме того, поставщик библиотеки может предоставить основную единицу интерфейса, объединяющую все «подмодули» через импорт и повторный экспорт их интерфейсов. В следующем примере мы переопределим модуль `deitel.math` из раздела 16.9, используя только основные единицы интерфейса, чтобы показать, какую гибкость это дает пользователям.

Основной интерфейс модуля deitel.math.powers

Для начала мы переименуем файл `deitel.math-powers.ixx` в `deitel.math.powers.ixx`. Ранее мы использовали условное обозначение «*-name*», указывая, что раздел `powers` был частью модуля `deitel.math`. В этой программе `deitel.math.powers` будет основной единицей интерфейса модуля (илл. 16.16). Вместо объявления раздела интерфейса, как на илл. 16.12:

```
export module deitel.math:powers;
```

---

[1] Ричард Смит (Richard Smith), «Merging Modules – Section 2.2 Module Partitions», 22 февраля 2019 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p1103r3.

[2] Стандарт C++, «10. Modules – 10.1. Module Units and Purviews».

в строке 3 на илл. 16.16 объявлена основная единица интерфейса, с точкой в имени:

```
export module deitel.math.powers;
```

Теперь мы можем независимо импортировать модуль `deitel.math.powers` и использовать его функции (илл. 16.17).

```
1 // Fig. 16.16: deitel.math.powers.ixx
2 // Основной интерфейс модуля deitel.math.powers
3 export module deitel.math.powers;
4
5 export namespace deitel::math {
6    double square(double x) {return x * x;}
7    double cube(double x) {return x * x * x;}
8 }
```

Илл. 16.16 | Основной интерфейс модуля deitel.math.powers

```
1 // fig16_17.cpp
2 // Использование функций модуля deitel.math.powers
3 import <iostream>;
4 import deitel.math.powers; // Импортируем модуль deitel.math.powers
5
6 using namespace deitel::math;
7
8 int main() {
9    std::cout << "square(6): " << square(6)
10       << "\ncube(5): " << cube(5) << '\n';
11 }
```

```
square(6): 36
cube(5): 125
```

Илл. 16.17 | Использование функций модуля deitel.math.powers

Компиляция примера в Visual C++

Добавьте файлы `deitel.math.powers.ixx` и `fig16_17.cpp` в проект Visual C++, как показано в разделе 16.6.1. Запустите проект, чтобы скомпилировать модуль и программу с функцией `main`.

Компиляция примера в g++

Выполните команды, приведенные в разделе 16.6.6, для компиляции заголовка стандартной библиотеки `<iostream>` в качестве заголовочной единицы. Затем скомпилируйте основную единицу интерфейса модуля:

```
g++ -fmodules-ts -c -x c++ deitel.math.powers.ixx
```

Скомпилируйте программу с функцией `main` и скомпонуйте ее с файлом `deitel.math.powers.o`:

```
g++ -fmodules-ts fig16_17.cpp deitel.math.powers.o -o fig16_17
```

Компиляция примера в clang++

Скомпилируйте основную единицу интерфейса модуля в файл `.pcm`:

```
clang++ -c -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -x c++ deitel.math.powers.ixx
   -Xclang -emit-module-interface -o deitel.math.powers.pcm
```

Скомпилируйте программу с функцией `main` и скомпонуйте ее с файлом `deitel.math.powers.o`:

```
clang++ -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -fprebuilt-module-path=.
   fig16_17.cpp deitel.math.powers.pcm -o fig16_17
```

Основная единица интерфейса модуля deitel.math.roots

Теперь мы переименуем файл `deitel.math-roots.ixx` в `deitel.math.roots.ixx` и сделаем `deitel.math.roots` основной единицей интерфейса модуля (илл. 16.18). Вместо объявления раздела, как показано на илл. 16.13:

```
export module deitel.math:roots;
```

в строке 3 на илл. 16.18 объявлена основная единица интерфейса модуля с точкой в имени:

```
export module deitel.math.roots;
```

Теперь мы можем независимо импортировать модуль `deitel.math.roots` и использовать его функции (илл. 16.19).

```
1 // Fig. 16.18: deitel.math.roots.ixx
2 // Основной интерфейс модуля deitel.math.roots
3 export module deitel.math.roots;
4
5 import <cmath>;
6
7 export namespace deitel::math {
8    double squareRoot(double x) {return std::sqrt(x);}
9    double cubeRoot(double x) {return std::cbrt(x);}
10 }
```

Илл. 16.18 | Основной интерфейс модуля deitel.math.roots

```
1 // fig16_19.cpp
2 // Использование функций модуля deitel.math.roots
3 import <iostream>;
4 import deitel.math.roots; // Импортируем модуль deitel.math.roots
5
6 using namespace deitel::math;
7
8 int main() {
```

```
 9    std::cout << "squareRoot(9): " << squareRoot(9)
10       << "\ncubeRoot(1000): " << cubeRoot(1000) << '\n';
11 }
```

```
squareRoot(9): 3
cubeRoot(1000): 10
```

Илл. 16.19 | Использование функций модуля deitel.math.roots

Компиляция примера в Visual C++

Добавьте файлы `deitel.math.roots.ixx` и `fig16_19.cpp` в проект Visual C++, как показано в разделе 16.6.1. Запустите проект, чтобы скомпилировать модуль и программу с функцией `main`.

Компиляция примера в g++

Выполните команды, приведенные в разделе 16.6.6, для компиляции заголовков стандартной библиотеки `<iostream>` и `<cmath>` в качестве заголовочных единиц. Затем скомпилируйте основную единицу интерфейса модуля:

```
g++ -fmodules-ts -c -x c++ deitel.math.powers.ixx
```

Скомпилируйте программу с функцией `main` и скомпонуйте ее с файлом `deitel.math.roots.o`:

```
g++ -fmodules-ts fig16_19.cpp deitel.math.powers.o -o fig16_19
```

Компиляция примера в clang++

Скомпилируйте основную единицу интерфейса модуля в файл `.pcm`:

```
clang++ -c -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -x c++ deitel.math.roots.ixx
   -Xclang -emit-module-interface -o deitel.math.roots.pcm
```

Скомпилируйте программу с функцией `main` и скомпонуйте ее с файлом `deitel.math.roots.o`:

```
clang++ -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -fprebuilt-module-path=.
   fig16_19.cpp deitel.math.roots.pcm -o fig16_19
```

Основная единица интерфейса модуля deitel.math

Программы, показанные на илл. 16.16 и 16.18, теперь являются отдельными модулями. Их имена `deitel.math.powers` и `deitel.math.roots` подразумевают логическую связь между ними, и оба модуля экспортируют пространство имен `deitel::math`, но они не определяют один и тот же модуль. Для удобства мы можем объединить эти отдельные модули в основную единицу интерфейса, которая экспортирует «подмодули», как показано в строках 7 и 8 на илл. 16.20. После этого мы сможем использовать все функции «подмодулей», импортировав `deitel.math` (илл. 16.21).

Благодаря нашим «подмодулям», разработчики получают выбор из нескольких вариантов:

- импортировать `deitel.math.powers`, когда нужны только функции `square` и `cube`;
- импортировать `deitel.math.roots`, когда нужны только функции `squareRoot` и `CubeRoot`;
- импортировать общий модуль `deitel.math`, когда нужны все четыре функции.

```
// fig. 16.20: deitel.math.ixx
// Основной интерфейс модуля deitel.math, объединяющий объявления из
// "подмодулей" deitel.math.powers и deitel.math.roots
export module deitel.math; // Основной интерфейс модуля

// Импортируем и реэкспортируем deitel.math.powers и deitel.math.roots
export import deitel.math.powers;
export import deitel.math.roots;
```

Илл. 16.20 | Основной интерфейс модуля deitel.math, объединяющий объявления из «подмодулей» deitel.math.powers и deitel.math.roots

```
// fig16_21.cpp
// Использование функций модуля deitel.math
import <iostream>;
import deitel.math; // Импортируем модуль deitel.math

using namespace deitel::math;

int main() {
   std::cout << "square(6): " << square(6)
      << "\ncube(5): " << cube(5)
      << "\nsquareRoot(9): " << squareRoot(9)
      << "\ncubeRoot(1000): " << cubeRoot(1000) << '\n';
}
```

```
square(6): 36
cube(5): 125
squareRoot(9): 3
cubeRoot(1000): 10
```

Илл. 16.21 | Использование функций модуля deitel.math

Компиляция примера в Visual C++

Добавьте файлы `.ixx` с илл. 16.16, 16.18 и 16.20, а также файл `fig16_21.cpp` в проект Visual C++, как показано в разделе 16.6.1. Запустите проект, чтобы скомпилировать модуль и программу с функцией `main`.

Компиляция примера в g++

Мы предполагаем, что вы выполняете команды в той же папке, в которой скомпилированы `deitel.math.powers.ixx` и `deitel.math.roots.ixx`. Скомпилируйте основную единицу интерфейса модуля:

```
g++ -fmodules-ts -c -x c++ deitel.math.ixx
```

Скомпилируйте программу с функцией `main` и скомпонуйте ее с файлами `deitel.math.powers.o`, `deitel.math.roots.o` и `deitel.math.o`:

```
g++ -fmodules-ts fig16_21.cpp deitel.math.powers.o
   deitel.math.roots.o deitel.math.o -o fig16_21
```

Компиляция примера в clang++

Мы предполагаем, что вы выполняете команды в той же папке, в которой скомпилированы `deitel.math.powers.ixx` и `deitel.math.roots.ixx`. Скомпилируйте основную единицу интерфейса модуля в файл `.pcm`:

```
clang++ -c -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -fprebuilt-module-path=.
   -x c++ deitel.math.ixx -Xclang -emit-module-interface
   -o deitel.math.pcm
```

Скомпилируйте программу с функцией `main` и скомпонуйте ее с файлами `deitel.math.powers.o`, `deitel.math.roots.o` и `deitel.math.o`:

```
clang++ -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -fprebuilt-module-path=. fig16_21.cpp
   deitel.math.pcm deitel.math.roots.pcm deitel.math.powers.pcm
   -o fig16_21
```

16.10. Дополнительные примеры работы с модулями

Следующие разделы главы 16 демонстрируют дополнительные концепции модулей:

- импорт модулей из стандартных библиотек Microsoft и clang++;
- некоторые ограничения модулей и ошибки компиляции, с которыми вы столкнетесь, если нарушите эти ограничения;
- различия между элементами модуля, которые другие единицы трансляции могут использовать по именам, и элементами модуля, которые другие единицы трансляции могут использовать косвенно.

16.10.1. Пример: импорт стандартной библиотеки C++ в виде модулей

В настоящее время стандарт C++ не требует, чтобы компиляторы предоставляли стандартную библиотеку с поддержкой модулей. Но корпорация Microsoft уже предоставляет ее для Visual C++, эта библиотека разделена на несколько модулей. Кроме того, clang++ предоставляет такую библиотеку в виде одного модуля. У компилятора g++ еще нет такой возможности на момент написания этой книги.

Вы можете импортировать следующие модули в проекты Visual C++ и «потенциально ускорить время компиляции в зависимости от размера вашего проекта»[1]:

- `std.core` — модуль с функциональностью большей части стандартной библиотеки (кроме перечисленных ниже элементов);
- `std.filesystem` — модуль с функциональностью заголовка `<filesystem>`;
- `std.memory` — модуль с функциональностью заголовка `<memory>`;
- `std.regex` — модуль с функциональностью заголовка `<regex>`;
- `std.threading` — модуль с функциональностью всех заголовков, относящихся к параллелизму: `<atomic>`, `<condition_variable>`, `<future>`, `<mutex>`, `<shared_mutex>` и `<thread>`.

В компиляторе clang++ вы можете импортировать всю стандартную библиотеку C++ одной инструкцией:

```
import std;
```

Существует предложение по двум стандартным библиотечным модулям с именами `std` и `std.compat`, которые, вероятно, будут введены в C++23[2].

В Visual C++ вы не можете комбинировать код, в который директивой `#include` включены заголовки стандартной библиотеки, и код, в который импортированы модули стандартной библиотеки Microsoft. Компилятор не будет знать, какую из двух версий стандартной библиотеки надо выбрать при компоновке исполняемого файла программы.

В программе, показанной на илл. 16.22, мы импортируем модуль `std.core` (строка 3), а затем выводим на экран сообщение с помощью объекта `cout`, который идентичен по функциональности объекту `cout` из заголовка `<iostream>` стандартной библиотеки.

---

[1] Колин Робертсон (Colin Robertson) и Ник Шоннинг (Nick Schonning), «Overview of Modules in C++», 13 декабря 2019 г. Был доступен: 15 сентября 2023 г. https://docs.microsoft.com/en-us/cpp/cpp/modules-cpp?view=msvc-160.

[2] Стефан Т. Лававей (Stephan T. Lavavej), Габриэль Дос Рейс (Gabriel Dos Reis), Бьерн Страуструп (Bjarne Stroustrup) и Джонатан Уэйкли (Jonathan Wakely), «Standard Library Modules std and std.compat», 13 октября 2021 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p2465.

```
1 // fig16_22.cpp
2 // Импорт стандартной библиотеки Microsoft в виде модулей
3 import std.core; // Основная часть стандартной библиотеки C++
4
5 int main() {
6    std::cout << "Welcome to C++20 Modules!\n";
7 }
```

```
Welcome to C++20 Modules!
```

Илл. 16.22 | Импорт модульной версии стандартной библиотеки C++ от Microsoft

Компиляция примера в Visual C++

Компиляция программы, использующей модули стандартной библиотеки Microsoft, требует дополнительных настроек проекта (они могут измениться, когда Microsoft завершит работу по интеграции модулей в Visual C++). Во-первых, вы должны убедиться, что установлена поддержка модулей C++:

1. В Visual Studio выберите `Tools` > `Get Tools and Features...`.
2. На вкладке `Individual Components` найдите пункт `C++ Modules` и убедитесь, что установлен флаг `C++ Modules for v### build tools` (на момент написания книги `###` — это `143`). Если флаг не установлен, установите его и нажмите `Modify` для установки. Вам нужно будет закрыть Visual Studio для завершения установки.

Далее вы должны настроить несколько параметров проекта:

1. В `Solution Explorer` щелкните правой кнопкой мыши на свой проект и выберите `Properties`, чтобы открыть диалоговое окно `Property Pages`.
2. В левой колонке выберите `Configuration Properties` > `C/C++` > `Code Generation`.
3. В правой колонке убедитесь, что для параметра `Enable C++ Exceptions` установлено значение `Yes (/EHsc)`.
4. В правой колонке убедитесь, что для `Runtime Library` установлено значение `Multi-threaded DLL (/MD)`, если вы компилируете в режиме `Release`, или `Multi-threaded Debug DLL (/MDd)`, если вы компилируете в режиме `Debug`. Вы можете выбрать настройки для режима `Release` или `Debug`, изменив значение в раскрывающемся списке `Configurations` в верхней части диалогового окна `Property Pages`.
5. В левой колонке выберите `Configuration Properties` > `C/C++` > `Language`.
6. В правой колонке убедитесь, что для параметра `Enable Experimental C++ Standard Library Modules` установлено значение `Yes (/experimental:module)`, а для стандарта языка C++ установлено значение `ISO C++20 Standard (/std:c++20)`.

Теперь вы можете скомпилировать и запустить код с илл. 16.22.

Модификация и компиляция программы в clang++

Для совместимости с компилятором clang++ замените инструкцию в строке 3 на илл. 16.22 следующей:

```
import std;
```

После этого скомпилируйте программу:

```
clang++ -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps fig16_22.cpp -o fig16_22
```

16.10.2. Пример: запрет циклических зависимостей

Модулю запрещено иметь циклическую зависимость, то есть модуль не может импортировать сам себя прямо или косвенно[1,2]. На илл. 16.23 и 16.24 определяются основные единицы интерфейса модулей `moduleA` и `moduleB`. Как видите, `moduleA` импортирует `moduleB`, и наоборот. Каждый модуль косвенно зависит от самого себя из-за инструкций `import` в строке 5 на илл. 16.23 и в строке 5 на илл. 16.24: `moduleA` импортирует `moduleB`, который, в свою очередь, импортирует модуль `moduleA`.

```
1 // Fig. 16.23: moduleA.ixx
2 // Основной интерфейс модуля, импортирующего moduleB
3 export module moduleA; // Объявляем основной интерфейс модуля
4
5 export import moduleB; // Импортируем и реэкспортируем moduleB
```

Илл. 16.23 | Основной интерфейс модуля, импортирующего moduleB

```
1 // Fig. 16.24: moduleB.ixx
2 // Основной интерфейс модуля, импортирующего moduleA
3 export module moduleB; // Объявляем основной интерфейс модуля
4
5 export import moduleA; // Импортируем и реэкспортируем moduleA
```

Илл. 16.24 | Основной интерфейс модуля, импортирующего moduleA

При попытке компиляции кода с илл. 16.23 и 16.24 Visual C++ выдает сообщение об ошибке:

```
error : Cannot build the following source files because there is a
cyclic dependency between them: ch16\fig16_23-24\moduleA.ixx depends
on ch16\fig16_23-24\moduleB.ixx depends on ch16\fig16_23-24\
moduleA.ixx.
```

---

[1] Стандарт C++, «10. Modules — 10.3. Import Declaration». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/module.import.

[2] «Understanding C++ Modules: Part 2: export, import, Visible, and Reachable», 31 марта 2019 г. Был доступен: 15 сентября 2023 г. https://vector-of-bool.github.io/2019/03/31/modules-2.html.

Вы не сможете скомпилировать этот пример ни в g++, ни в clang++. Оба компилятора требуют, чтобы основной интерфейс модуля был скомпилирован, прежде чем вы сможете его импортировать. Поскольку модули зависят друг от друга, это невозможно.

16.10.3. Пример: инструкции import не наследуются

В разделе 16.6.7 вы узнали, что модули имеют жесткую инкапсуляцию и не экспортируют объявления неявным образом. Это означает, что инструкции `import` не наследуются: если единица трансляции импортирует `A`, а `A` импортирует `B`, то та единица трансляции, которая импортирует `A`, не получит автоматически доступ к экспортированным элементам `B`.

Рассмотрим `moduleA` (илл. 16.25) и `moduleB` (илл. 16.26): `moduleB` импортирует, но не экспортирует `moduleA` (строка 6 на илл. 16.26). В результате функция `cube` модуля `moduleA` не является частью интерфейса `moduleB`.

```
1 // Fig. 16.25: moduleA.ixx
2 // Основной интерфейс модуля, экспортирующего функцию cube
3 export module moduleA; // Объявляем основной интерфейс модуля
4
5 export int cube(int x) { return x * x * x; }
```

Илл. 16.25 | Основной интерфейс модуля, экспортирующего функцию cube

```
1 // Fig. 16.26: moduleB.ixx
2 // Основной интерфейс модуля, который импортирует (но не экспортирует)
3 // модуль moduleA и экспортирует функцию square
4 export module moduleB; // Объявляем основной интерфейс модуля
5
6 import moduleA; // Импортируем, но не экспортируем moduleA
7
8 export int square(int x) { return x * x; }
```

Илл. 16.26 | Основной интерфейс модуля, который импортирует (но не экспортирует) модуль moduleA и экспортирует функцию square

В программе, показанной на илл. 16.27, импортируется `moduleB` и осуществляется попытка вызвать функцию `cube` модуля `moduleA` (строка 8). Это вызывает ошибки, поскольку объявление функции `cube` не импортировано. Приведем основные сообщения об ошибках, которые выдают Visual C++, g++ и clang++ соответственно:

- `error C3861: 'cube': identifier not found`
- `error: 'cube' was not declared in this scope`
- `error: declaration of 'cube' must be imported from module 'moduleA' before it is required`

```
1 // fig16_27.cpp
2 // Модуль moduleB не экспортирует неявно функции модуля moduleA
3 import <iostream>;
4 import moduleB;
5
6 int main() {
7    std::cout << "square(6): " << square(6) // Экспорт из модуля moduleB
8       << "\ncube(5): " << cube(5) << '\n'; // Нет экспорта из модуля moduleB
9 }
```

Илл. 16.27 | Модуль moduleB не экспортирует неявно функции модуля moduleA

16.10.4. Пример: видимость и доступность модулей

Каждый пример, в котором мы использовали экспорт имени, демонстрировал концепцию **видимости (visibility)**. Объявление «видно» в единице трансляции, если вы можете использовать его имя. Как было показано, любое экспортированное из модуля имя можно использовать в единицах трансляции, импортирующих этот модуль.

Некоторые объявления **доступны (reachable)**, но не видны. Это означает, что вы не можете явно указать имя объявления в другой единице трансляции, но объявление доступно косвенно[1,2,3]. Все видимое доступно, но не наоборот. Концепцию доступности легче всего объяснить на примере кода. Чтобы продемонстрировать доступность, мы модифицировали основную единицу интерфейса модуля `deitel.time.ixx`[4] с илл. 16.9. Вот два ключевых изменения (илл. 16.28):

- Мы больше не экспортируем пространство имен `deitel::time` (строка 7), поэтому класс `Time` не экспортируется. Следовательно, он не виден единицам трансляции, импортирующим `deitel.time`.
- Мы экспортировали функцию `getTime` (строка 21), возвращающую объект `Time` в точку вызова. Мы будем использовать возвращаемое значение, демонстрируя концепцию доступности.

Единица реализации модуля (илл. 16.10) в этом примере остается неизменной, поэтому мы ее не дублируем.

```
1 // Fig. 16.28: deitel.time.ixx
2 // Основной интерфейс модуля deitel.time
3 export module deitel.time; // Объявляем основной интерфейс модуля
4
```

[1] Стандарт C++, «10. Modules — 10.7. Reachability». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/module.reach.

[2] «Understanding C++ Modules: Part 2: export, import, Visible, and Reachable».

[3] Ричард Смит (Richard Smith), «Merging Modules — Section 2.2 Module Partitions», 22 февраля 2019 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p1103r3.

[4] На момент написания книги этот пример неправильно компоновался в g++ и не компилировался в clang++.

```
5 import <string>; // Вместо #include <string>
6
7 namespace deitel::time {
8    class Time { // Не экспортируется
9    public:
10      // Конструктор по умолчанию, так как он может быть вызван без аргументов
11      explicit Time(int hour = 0, int minute = 0, int second = 0);
12
13      std::string toString() const;
14   private:
15      int m_hour{0}; // 0 - 23 (24-часовой формат)
16      int m_minute{0}; // 0 - 59
17      int m_second{0}; // 0 - 59
18   };
19
20   // Экспортированная функция возвращает объект Time
21   export Time getTime() {return Time(6, 45, 0);}
22 }
```

Илл. 16.28 | Основной интерфейс модуля deitel.time

Показанная на илл. 16.29 программа импортирует модуль **deitel.time** (строка 5). В строке 10 вызывается функция модуля **getTime** для получения объекта **Time**. Обратите внимание, что тип переменной **t** определяется автоматически. Если заменить **auto** в строке 10 на **deitel::time::Time**, вы получите ошибку примерно такого вида (в Visual C++):

```
'Time': is not a member of 'deitel::time'
```

Ошибка возникает потому, что класс **Time** не виден в этой единице трансляции. Однако определение класса доступно, потому что функция **getTime** возвращает объект **Time**. Компилятор это знает и может определить тип переменной **t**. Когда определение класса доступно, элементы класса становятся видимыми. Таким образом, даже если модуль **deitel.time** не экспортирует класс **Time**, единица трансляции может вызывать функцию **toString** (строка 14) класса **Time**, чтобы получить строковое представление объекта **t**. Этапы компиляции этой программы такие же, как в разделе 16.8.2, за исключением имени файла основной программы, который теперь называется **fig16_29.cpp**.

```
1 // fig16_29.cpp
2 // Демонстрация доступности типа deitel::time::Time
3 // и видимости его открытых элементов данных
4 import <iostream>;
5 import deitel.time;
6
7 int main() {
8    // Инициализируем t объектом, возвращенным функцией getTime; нельзя
9    // объявить t с типом Time, так как этот тип не экспортирован, т.е. не виден
10   auto t{deitel::time::getTime()};
11
```

Окончание ⇨

```
12    // Функция toString класса Time доступна, даже если
13    // класс Time не экспортирован модулем deitel.time
14    std::cout << "Time t:\n" << t.toString() << "\n\n";
15 }
```

```
Time t:
Hour: 6
Minute: 45
Second: 0
```

Илл. 16.29 | Демонстрация доступности типа deitel::time::Time и видимости его открытых элементов данных

16.11. Перенос кода в модули

Мы часто обращались к C++ Core Guidelines за советами и рекомендациями по использованию различных языковых элементов. На момент написания этой книги технология модулей все еще считается новинкой, а реализация модулей в популярных компиляторах не завершена. Даже в C++ Core Guidelines пока нет рекомендаций по работе с модулями. Все еще мало статей и видеороликов, в которых обсуждается опыт разработчиков по переносу существующих программных систем в модули. Мы перечислим несколько публикаций, которые нам понравились. Видеоролики Кэмерона Да Камара (Microsoft) и Стива Дауни (Bloomberg) содержат свежие советы, рекомендации и аналитические материалы. В видеороликах Даниэлы Энгерт и Натана Сидуэлла демонстрируется перенос существующего кода в модули, а в статье Юки Такахаси, Оксаны Шадуры и Васила Василева обсуждается их опыт работы с модулями в большой кодовой базе CERN ROOT C++:

- Кэмерон Да Камара (Cameron DaCamara), «Moving a Project to C++ Named Modules», 10 августа 2021 г. Был доступен: 15 сентября 2023 г. https://devblogs.microsoft.com/cppblog/moving-a-project-to-cpp-named-modules/.
- Стив Дауни (Steve Downey), «Writing a C++20 Module», 5 июля 2021 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=AO4piAqV9mg.
- Даниэла Энгерт (Daniela Engert), «Modules: The Beginner's Guide», 2 мая 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=Kqo-jIq4V3I.
- Юки Такахаси (Yuka Takahashi), Оксана Шадура (Oksana Shadura) и Васил Василев (Vassil Vassilev), «Migrating Large Codebases to C++ Modules», 22 августа 2019 г. Был доступен: 15 сентября 2023 г. https://arxiv.org/abs/1906.05092.
- Натан Сидуэлл (Nathan Sidwell), «Converting to C++20 Modules», 4 октября 2019 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=KVsWIEw3TTw.

Мы будем размещать дополнительные ресурсы по мере их поступления по адресу:

https://deitel.com/c-plus-plus-20-for-programmers

16.12. Будущее модулей и инструментов для работы с ними

Комитет по стандартам C++ начал свою работу над C++23, и одним из ключевых элементов этой работы является модульная стандартная библиотека[1]. Модули C++20 еще настолько новы, что инструменты, помогающие их использовать, находятся в стадии разработки и будут развиваться в течение нескольких лет. Но с некоторыми из них вы уже знакомы. Например, если вы изучали примеры этой главы в Visual C++, вы могли заметить, что Visual Studio уже позволяет добавлять модули в ваши проекты, а имеющиеся в Visual Studio инструменты сборки могут компилировать и компоновать ваши модули и приложения.

Во многих других популярных языках программирования тоже появились системы модулей или аналогичные возможности:

- В Java есть Java Platform Module System (JPMS), о которой мы написали главу в наших книгах о Java и опубликовали статью в Oracle Java Magazine[2].
- В Python есть хорошо развитая система модулей, которую мы много использовали в наших книгах об этом языке[3, 4].
- В языках платформы Microsoft .NET, таких как C# и Visual Basic, поддерживается модульная система на основе **сборок (assemblies)**.

В «Википедии» перечислены десятки языков, имеющих те или иные модульные системы[5]. Многие языки предоставляют инструменты, которые помогают работать с их модульными системами и преобразовывать имеющийся код в модули. Следующие инструменты могут со временем появиться в экосистеме C++:

- средства сборки с поддержкой модулей, которые управляют компиляцией программных систем (в Visual C++ это уже есть);
- инструменты для создания кросс-платформенных интерфейсов модулей, позволяющие разработчикам распространять описания интерфейсов и объектный код вместо исходного кода;
- инструменты проверки зависимостей, позволяющие убедиться, что все необходимые модули установлены;

---

1 «To Boldly Suggest an Overall Plan for C++23», 25 ноября 2019 г. https://wg21.link/p0592r4.

2 Пол Дейтел (Paul Deitel), «Understanding Java 9 Modules», Oracle Java Magazine, сентябрь-октябрь 2017 г. https://www.oracle.com/a/ocom/docs/corporate/java-magazine-sept-oct-2017.pdf.

3 Пол Дейтел (Paul Deitel) и Харви Дейтел (Harvey Deitel), «Python for Programmers», 2019. Изд. Pearson Education.

4 Пол Дейтел (Paul Deitel) и Харви Дейтел (Harvey Deitel), «Intro to Python for Computer Science and Data Science: Learning to Program with AI, Big Data and the Cloud», 2020. Изд. Pearson Education.

5 «Modular Programming». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Modular_programming.

- инструменты для сбора статистики, определяющие, какие модули и каких версий установлены;
- инструменты, которые визуализируют зависимости модулей, показывая вам взаимосвязи между модулями в программных системах;
- инструменты упаковки и распространения модулей, помогающие разработчикам удобно устанавливать модули и их зависимости на разных платформах;

и многое другое.

Рекомендации

Статья Даниэля Руоссо из Bloomberg[1], опубликованная в июле 2021 года, рассматривает ряд проблем повторного использования кода и сборки больших приложений, стимулируя дискуссии о будущем модулей C++. В этой статье и в других ресурсах, указанных ниже, перечислены в обратном хронологическом порядке полезные возможности и средства поддержки модулей (как в рамках стандарта C++, так и от сторонних поставщиков), улучшающие процесс разработки программ на C++:

- Даниэль Руоссо (Daniel Ruoso), «Requirements for Usage of C++ Modules at Bloomberg», 12 июля 2021 г. Был доступен: 15 сентября 2023 г. https://isocpp.org/files/papers/P2409R0.pdf.
- Натан Сидуэлл (Nathan Sidwell), «P1184: A Module Mapper», 10 июля 2020 г. Был доступен: 15 сентября 2023 г. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1184r2.pdf.
- Роб Ирвинг (Rob Irving), Джейсон Тернер (Jason Turner) и Габриэль Дос Рейс (Gabriel Dos Reis), «Modules Present and Future», 18 июня 2020 г. Был доступен: 15 сентября 2023 г. https://cppcast.com/modules-gaby-dos-reis.
- Кэмерон Да Камара (Cameron DaCamara), «Practical C++20 Modules and the Future of Tooling Around C++ Modules», 4 мая 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=ow2zV0Udd9M.
- Натан Сидуэлл (Nathan Sidwell), «C++ Modules and Tooling», 4 октября 2018 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=4yOZ8Zp_Zfk.
- Габриэль Дос Рейс (Gabriel Dos Reis), «Modules Are a Tooling Opportunity», 16 октября 2017 г. Был доступен: 15 сентября 2023 г. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0822r0.pdf.

16.13. Итоги

В этой главе мы рассмотрели модули — перспективную технологию из «большой четверки» новинок C++20. Вы увидели, что модули помогают вам структурировать

[1] Даниэль Руоссо (Daniel Ruoso), «Requirements for Usage of C++ Modules at Bloomberg», 12 июля 2021 г. Был доступен: 15 сентября 2023 г. https://wg21.link/P2409R0.

код, контролировать, какие объявления вы предоставляете клиентскому коду, и инкапсулировать детали реализации. Мы рассмотрели преимущества модулей и объяснили, как они повышают продуктивность разработчиков, упрощают масштабирование систем, сокращают размеры единиц трансляции и время сборки. Мы также отметили ряд их недостатков.

Мы рассмотрели ряд примеров кода с модулями. Вы узнали, как можно улучшить даже небольшие программы с помощью технологии модулей, перейдя от директив препроцессора `#include` к импорту заголовков стандартной библиотеки в виде заголовочных единиц. Вы научились создавать пользовательские модули. Вы научились определять основную единицу интерфейса модуля для спецификации интерфейса клиентского кода, а также импортировать модуль в приложение, получая в свое распоряжение его экспортированные элементы. Вы использовали пространства имен, избегая конфликтов имен с другими модулями. Вы узнали, что неэкспортированные элементы модуля недоступны по имени в импортирующих единицах трансляции.

Вы научились отделять интерфейс от реализации — сначала в основном интерфейсе модуля, помещая код реализации во фрагмент `:private`, а затем в отдельных файлах единиц реализации. Вы разделили модуль на разделы, структурируя его компоненты в виде маленьких, хорошо управляемых единиц трансляции. Вы узнали, что «подмодули» дают вам больше гибкости, чем разделы, поскольку нельзя импортировать разделы в сторонние единицы трансляции, не входящие в модуль.

Мы показали, что модульная версия стандартной библиотеки C++ от Microsoft или от clang++ импортируется очень просто: всего одной инструкцией. Вы узнали о запрете циклических зависимостей в модулях и об отсутствии наследования при импорте. Мы рассмотрели разницу между видимостью и доступностью, упомянув, что все видимое доступно, но все доступное не обязательно является видимым.

Мы предоставили ресурсы с советами по переносу устаревшего кода в модули — эта тема очень интересна для организаций, рассматривающих возможность использования модульной технологии. Наконец, мы обсудили ближайшее будущее модулей и сопутствующий инструментарий, который может быть реализован в ближайшие несколько лет. На следующих страницах мы предоставляем вам для дальнейшего самостоятельного изучения списки видеороликов, статей и технической документации, к которым мы обращались, работая над этой главой. Далее мы приводим краткий глоссарий по модулям с ключевыми терминами и определениями.

Модульная технология крайне важна. С правильным инструментарием модули предоставят разработчикам C++ большие возможности для улучшения процессов проектирования, реализации и эволюции библиотек и крупномасштабных программных систем.

Но модули все еще являются очень новой технологией, поэтому пока лишь немногие организации пробовали работать с ними. Некоторые организации, конечно, будут использовать модули при создании нового программного обеспечения. Но как быть со старыми программами, написанными на C++ за четыре десятилетия?

Какие-то из них рано или поздно будут переписаны по модульной технологии, но многие — никогда, в том числе из-за того, что из ИТ-компаний давно ушли их создатели, хорошо понимающие их код.

В Deitel & Associates мы работаем с распространенными языками программирования. Основываясь на нашем опыте изучения, написания и преподавания модульной системы Java Platform Module System (JPMS), мы считаем, что внедрение модулей C++20 будет постепенным. Java представила JPMS в 2017 году. Опрос программистов, проведенный в 2021 году поставщиком компиляторов JetBrains, показал, что 72% из них все еще работают с версией Java 8 (2014), в которой нет поддержки JPMS[1]. Организации, использующие C++, скорее всего, тоже будут действовать осторожно. Многие предпочтут подождать, чтобы сначала изучить чужой опыт переноса больших кодовых баз на модульную технологию и запуска новых проектов на основе модулей.

В следующей главе мы представим вам имеющиеся в C++ технологии поддержки мощных многоядерных архитектур современных компьютеров: конкурентность, параллелизм и параллельные алгоритмы стандартной библиотеки.

Приложение: видео про модули

- Габриэль Дос Рейс (Gabriel Dos Reis) и Камерон Да Камара (Cameron DaCamara), «Implementing C++ Modules: Lessons Learned, Lessons Abandoned», 18 декабря 2021 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=9OWGgkuyFV8.
- Стив Дауни (Steve Downey), «Writing a C++20 Module», 5 июля 2021 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=AO4piAqV9mg.
- Даниэла Энгерт (Daniela Engert), «The Three Secret Spices of C++ Modules», 1 июля 2021 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=l_83lyxWGtE.
- Сай Бренд (Sy Brand), «C++ Modules: Year 2021», 6 мая 2021 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=YcZntyWpqVQ.
- Габриэль Дос Рейс, «Programming in the Large with C++20: Meeting C++ 2020 Keynote», 11 декабря 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=j4du4LNsLiI.
- Марк Грегуар (Marc Gregoire), «C++20: An (Almost) Complete Overview», 26 сентября 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=FRkJCvHWdwQ.
- Кэмерон Да Камара, «Practical C++20 Modules and the Future of Tooling Around C++ Modules», 4 мая 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=ow2zV0Udd9M.

[1] «The State of Developer Ecosystem 2021», 15 июля 2021 г. Был доступен: 15 сентября 2023 г. https://www.jetbrains.com/lp/devecosystem-2021/.

- Тимур Думлер (Timur Doumler), «How C++20 Changes the Way We Write Code», 10 октября 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=ImLFlLjSveM.
- Даниэла Энгерт, «Modules: The Beginner's Guide», 2 мая 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=Kqo-j1q4V3I.
- Брайс Адельштейн Лельбах (Bryce Adelstein Lelbach), «Modules Are Coming», 1 мая 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=yee9i2rUF3s.
- Конференция «Pure Virtual C++ 2020», 30 апреля 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=c1ThUFISDF4.
- «Demo: C++20 Modules», 30 марта 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=6SKIUeRaLZE.
- Даниэла Энгерт, «Dr Module and Sister #include», 5 декабря 2019 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=OCFOTle2G-A.
- Борис Колпаков, «Practical C++ Modules», 18 октября 2019 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=szHV6RdQdg8.
- Майкл Спенсер (Michael Spencer), «Building Modules», 6 октября 2019 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=L0SHHkBenss.
- Габриэль Дос Рейс, «Programming with C++ Modules: Guide for the Working Software Developer», 5 октября 2019 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=tjSu-KOz5HK4.
- Натан Сидуэлл (Nathan Sidwell), «Converting to C++20 Modules», 4 октября 2019 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=KVsWIEw3TTw.
- Габриэль Дос Рейс, «C++ Modules: What You Should Know», 13 сентября 2019 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=MP6SJEBt6Ss.
- Ричард Салай (Richárd Szalay), «The Rough Road Towards Upgrading to C++ Modules», 16 июня 2019 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=XJxQs8qgn-c.
- Натан Сидуэлл, «C++ Modules and Tooling», 4 октября 2018 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=4yOZ8Zp_Zfk.

Приложение: статьи про модули

Статьи перечислены в обратном хронологическом порядке:

- Кэмерон Да Камара, «Moving a Project to C++ Named Modules», 10 августа 2021 г. Был доступен: 15 сентября 2023 г. https://devblogs.microsoft.com/cppblog/moving-a-project-to-cppnamed-modules/.

- Кэмерон ДаКамара, «Using C++ Modules in MSVC from the Command Line Part 1: Primary Module Interfaces», 21 июля 2021 г. Был доступен: 15 сентября 2023 г. https://devblogs.microsoft.com/cppblog/using-cpp-modules-in-msvc-from-the-command-line-part-1/.
- Даниэль Руосо (Daniel Ruoso), «Requirements for Usage of C++ Modules at Bloomberg», 12 июля 2021 г. Был доступен: 15 сентября 2023 г. https://wg21.link/P2409R0.
- Андреас Фертиг (Andreas Fertig), «Programming with C++20: Concepts, Coroutines, Ranges, and More», 2021. https://andreasfertig.info/books/programming-with-cpp20/.
- Натан Сидуэлл, «C++ Modules: A Brief Tour», 19 октября 2020 г. Был доступен: 15 сентября 2023 г. https://accu.org/journals/overload/28/159/sidwell/.
- Кэмерон Да Камара, «Standard C++20 Modules Support with MSVC in Visual Studio 2019 Version 16.8», 14 сентября 2020 г. Был доступен: 15 сентября 2023 г. https://devblogs.microsoft.com/cppblog/standard-c20-modules-support-with-msvc-in-visual-studio-2019-version-16-8/.
- Васил Василев (Vassil Vassilev), Дэвид Ланге (David Lange), Малик Шахзад Музаффар (Malik Shahzad Muzaffar), Мирчо Родозов (Mircho Rodozov), Оксана Шадура и Александр Пенев, «C++ Modules in ROOT and Beyond», 25 августа 2020 г. Был доступен: 15 сентября 2023 г. https://arxiv.org/pdf/2004.06507.pdf.
- Бьерн Страуструп (Bjarne Stroustrup), «Thriving in a Crowded and Changing World: C++ 2006–2020 — Section 9.3.1 Modules», 12 июня 2020 г. Был доступен: 15 сентября 2023 г. https://www.stroustrup.com/hopl20main-p5-p-bfc9cd4--final.pdf.
- Райнер Гримм (Rainer Grimm), «C++20: Further Open Questions to Modules», 8 июня 2020 г. Был доступен: 15 сентября 2023 г. https://www.modernescpp.com/index.php/c-20-open-questions-to-modules.
- Райнер Гримм, «C++20: Structure Modules», 1 июня 2020 г. Был доступен: 15 сентября 2023 г. https://www.modernescpp.com/index.php/c-20-divide-modules.
- Райнер Гримм, «C++20: Module Interface Unit and Module Implementation Unit», 25 мая 2020 г. Был доступен: 15 сентября 2023 г. https://www.modernescpp.com/index.php/c-20-moduleinterface-unit-and-module-implementation-unit.
- Райнер Гримм, «C++20: A Simple Math Module», 17 мая 2020 г. Был доступен: 15 сентября 2023 г. https://www.modernescpp.com/index.php/cpp20-a-first-module.
- Корентен Жабо (Corentin Jabot), «What Do We Want from a Modularized Standard Library?», 16 мая 2020 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p2172r0.
- Райнер Гримм, «C++20: The Advantages of Modules», 10 мая 2020 г. Был доступен: 15 сентября 2023 г. https://www.modernescpp.com/index.php/cpp20-modules.

- Кэмерон Да Камара, «C++ Modules Conformance Improvements with MSVC in Visual Studio 2019 16.5», 22 января 2020 г. Был доступен: 15 сентября 2023 г. https://devblogs.microsoft.com/cppblog/c-modules-conformance-improvements-with-msvc-in-visual-studio-2019-16-5/.
- Колин Робертсон (Colin Robertson) и Ник Шоннинг (Nick Schonning), «Overview of Modules in C++», 13 декабря 2019 г. Был доступен: 15 сентября 2023 г. https://docs.microsoft.com/en-us/cpp/cpp/modules-cpp?view=msvc-160.
- Артур О' Двайр (Arthur O'Dwyer), «Hello World with C++2a Modules», 7 ноября 2019 г. Был доступен: 15 сентября 2023 г. https://quuxplusone.github.io/blog/2019/11/07/modular-hello-world/.
- «Understanding C++ Modules: Part 3: Linkage and Fragments», 7 октября 2019 г. Был доступен: 15 сентября 2023 г. https://vector-of-bool.github.io/2019/10/07/modules-3.html.
- Райнер Гримм, «More Details to Modules», 13 мая 2019 г. Был доступен: 15 сентября 2023 г. http://modernescpp.com/index.php/c-20-more-details-to-modules.
- Райнер Гримм, «Modules», 6 мая 2019 г. Был доступен: 15 сентября 2023 г. http://modernescpp.com/index.php/c-20-modules.
- Брайс Адельштейн Лельбах и Бен Крейг, «P1687R1: Summary of the Tooling Study Group's Modules Ecosystem Technical Report Telecons», 5 августа 2019 г. Был доступен: 15 сентября 2023 г. https://wg21.link/P1687R1.
- Корентен Жабо, «Naming Guidelines for Modules», 16 июня 2019 г. Был доступен: 15 сентября 2023 г. https://wg21.link/P1634R0.
- «Understanding C++ Modules: Part 2: export, import, Visible, and Reachable», 31 марта 2019 г. Был доступен: 15 сентября 2023 г. https://vector-of-bool.github.io/2019/03/31/modules-2.html.
- «Understanding C++ Modules: Part 1: Hello Modules, and Module Units», 10 марта 2019 г. Был доступен: 15 сентября 2023 г. https://vector-of-bool.github.io/2019/03/10/modules-1.html.
- Рене Ривера (Rene Rivera), «Are Modules Fast? (Revision 1)», 6 марта 2019 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p1441r1.
- Ричард Смит (Richard Smith), «Merging Modules», 22 февраля 2019 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p1103r3.
- «C++ Modules Might Be Dead-on-Arrival», 27 января 2019 г. Был доступен: 15 сентября 2023 г. https://vector-of-bool.github.io/2019/01/27/modules-doa.html.
- Ричард Смит и Габриэль Дос Рейс, «Merging Modules», 22 июня 2018 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p1103r0.
- Габриэль Дос Рейс и Ричард Смит, «Modules for Standard C++», 7 мая 2018 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p1087r0.

- Ричард Смит, «Another Take on Modules (Revision 1)», 6 марта 2018 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p0947r1.
- Бьерн Страуструп, «Modules and Macros», 11 февраля 2018 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p0955r0.
- Дмитрий Гузеев, «A Few Words on C++ Modules», 8 января 2018 г. Был доступен: 15 сентября 2023 г. https://medium.com/@dmitrygz/brief-article-on-c-modules-f58287a6c64.
- Габриэль Дос Рейс (ред.), «Working Draft, Extensions to C++ for Modules», 29 января 2018 г. Был доступен: 15 сентября 2023 г. https://wg21.link/n4720.
- Габриэль Дос Рейс и Павел Кертис (Pavel Curtis), «Modules, Componentization, and Transition», 5 октября 2015 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p0141r0.
- Габриэль Дос Рейс, Марк Холл (Mark Hall) и Гор Нишанов (Gor Nishanov), «A Module System for C++», 27 мая 2014 г. Был доступен: 15 сентября 2023 г. https://wg21.link/n4047.
- Давид Вандевурд (Daveed Vandevoorde), «Modules in C++ (Revision 6)», 11 января 2012 г. http://www.openstd.org/jtc1/sc22/wg21/docs/papers/2012/n3347.pdf.

Документация

- «3.23. C++ Modules». Был доступен: 15 сентября 2023 г. https://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Modules.html.
- Стандарт C++20, «10 Modules». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/module.
- «Modules — Module Partitions». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/language/modules.

Приложение: глоссарий по модулям

- **Видимое объявление (visible declaration)** — объявление, которое вы можете использовать по имени в вашем коде. Например, если вы импортируете модуль в единицу трансляции, экспортированные объявления модуля будут видны в этой единице трансляции.
- **Глобальный модуль (global module)** — неименованный модуль, содержащий все идентификаторы, которые определены в немодульных единицах трансляции или в глобальных фрагментах.
- **Глобальный фрагмент модуля (global module fragment)** — в модульной единице этот фрагмент может содержать только директивы препроцессора и должен находиться перед объявлением модуля. Все объявления в этом фрагменте являются частью глобального модуля и могут использоваться во всей остальной части модульной единицы.

- **Доступное объявление (reachable declaration)** — объявление является доступным, если вы можете использовать его в коде без прямой ссылки на него. Например, если вы импортируете модуль в единицу трансляции и одна из экспортированных функций модуля возвращает объект неэкспортируемого типа, этот тип доступен в импортирующих единицах трансляции.
- **Единица интерфейса модуля (module interface unit)** — модульная единица, в которой объявление модуля начинается с ключевого слова `export`.
- **Единица реализации модуля (module implementation unit)** — модульная единица, в которой объявление модуля не начинается с ключевого слова `export`.
- **Единица трансляции (translation unit)** — обработанный препроцессором файл исходного кода, готовый к компиляции.
- **Зависимость от интерфейса (interface dependency)** — если вы импортируете модуль в единицу реализации, то эта единица реализации становится зависимой от интерфейса модуля.
- **Заголовочная единица (header unit)** — заголовок, который импортируется инструкцией `import`, а не включается директивой препроцессора `#include`.
- **Именованная область действия модуля (named module purview)** — области действия всех модульных единиц в именованном модуле.
- **Именованный модуль (named module)** — все модульные единицы с одним и тем же именем модуля.
- **Импорт (`import`) заголовочного файла** — позволяет обрабатывать существующие заголовки как заголовочные единицы, что может сокращать время компиляции в больших проектах.
- **Импортировать (`import`) модуль** — сделать экспортируемые декларации модуля доступными в единице трансляции.
- **Имя модуля (module name)** — имя, указанное в объявлении модуля. Все модульные единицы одного и того же модуля должны иметь одинаковое имя модуля.
- **Модульная единица (module unit)** — единица реализации, содержащая объявление модуля.
- **Модульное связывание (module linkage)** — имена в модуле, экспортируемые из него, известны только в этом модуле.
- **Область действия модуля (module purview)** — множество идентификаторов внутри модульной единицы, начиная с объявления модуля и до конца единицы трансляции.
- **Объявление `import` (`import` declaration)** — инструкция, импортирующая модуль в единицу трансляции.
- **Объявление модуля (module declaration)** — каждая модульная единица имеет объявление модуля, в котором указано имя этого модуля и, возможно, имя раздела.

- **Основная единица интерфейса модуля (primary module interface unit)** — определяет множество деклараций, экспортируемых модулем для использования в других единицах трансляции.
- **Пространство имен (namespace)** — определяет область видимости, в которой находятся идентификаторы и переменные, что помогает предотвратить конфликты имен между идентификаторами в ваших собственных программах и библиотеках.
- **Раздел (partition)** — разновидность модуля, определяющая часть интерфейса или реализации модуля. Разделы не видны единицам трансляции, импортирующим модуль.
- **Раздел модуля (module partition)** — модульная единица, содержащая в объявлении модуля имя модуля, за которым следует двоеточие и имя раздела. Имена разделов модуля в одном и том же именованном модуле должны быть уникальными. Если раздел модуля является интерфейсным, он должен быть экспортирован основной единицей интерфейса этого модуля.
- **Формат IFC (.ifc)** — формат файлов Microsoft Visual C++, содержащих информацию, которую компилятор генерирует для модуля.
- **Формат PCM (.pcm — precompiled module interface)** — формат файлов clang++, содержащих информацию об интерфейсе модуля. Используется при компиляции других единиц трансляции, зависящих от данного модуля.
- **Фрагмент** `:private` — секция в основной единице интерфейса модуля, позволяющая определить реализацию модуля в одном файле с его интерфейсом, не раскрывая эту реализацию другим единицам трансляции.
- **Экспорт объявления модуля** (`export` **module declaration**) — указывает, что модульная единица является основным интерфейсом модуля, и вводит имя этого модуля.
- **Экспорт блока** (`export {}`) — модуль экспортирует все объявления или определения, содержащиеся в блоке (в фигурных скобках). Блок не определяет новую область видимости.
- **Экспортировать объявление** (`export` **a declaration**) — сделать объявление доступным для единиц трансляции, импортирующих соответствующий модуль.
- **Экспортировать определение** (`export` **a definition**) — сделать определение (например, шаблон) доступным для единиц трансляции, импортирующих соответствующий модуль.

17

Параллельные алгоритмы и конкурентность: высокоуровневый подход

В этой главе вы:

- познакомитесь с понятиями параллелизма, конкурентности и многопоточности;
- начнете использовать высокоуровневые инструменты конкурентного выполнения задач: параллельные алгоритмы (C++17), а также защелки и барьеры (C++20);
- поймете, что такое жизненный цикл потока;
- используете заголовок <chrono>, чтобы измерять быстродействие последовательных и параллельных алгоритмов в многоядерных системах;
- научитесь реализовывать правильное взаимодействие между производителем и потребителем;
- синхронизируете доступ нескольких потоков к общим изменяемым данным, используя классы `std::mutex`, `std::lock_guard`, `std::condition_variable` и `std::unique_lock`;
- используете шаблон функции `std::async` и шаблон класса `std::future` для асинхронного выполнения длительных вычислений и для получения их результатов;
- примените инструменты конкурентного выполнения задач (C++20), включая защелки, барьеры, семафоры и атомарные типы;
- узнаете о перспективных инструментах конкурентного программирования, которые могут появиться в будущих версиях C++.

17.1. Введение

Хорошо, когда есть возможность сосредоточиться на какой-то одной задаче и добросовестно выполнять ее, не отвлекаясь ни на что другое, пока она не будет завершена. Но это не всегда удается в сложном мире, где так много всего происходит одновременно. В этой главе представлены имеющиеся в C++ инструменты для создания программ, содержащих конкурентные задачи. Такой подход может значительно повышать быстродействие и скорость реакции программ.

Последовательное, конкурентное и параллельное выполнение задач

Когда мы говорим, что две задачи выполняются **последовательно (sequentially)**, мы имеем в виду их строгую очередность: сначала завершается первая, и только потом начинается вторая. Когда мы говорим, что две задачи выполняются **конкурентно (concurrently)**, мы имеем в виду, что они обе находятся в процессе выполнения: возможно, продвигаясь маленькими шагами и не всегда синхронно.

Давайте уточним. До начала 2000-х годов в большинстве компьютеров был одноядерный процессор. Операционная система на таком компьютере выполняет задачи конкурентно, быстро переключаясь между ними: завершает маленький шаг первой задачи, переходит к шагу второй задачи, и т. д. В каждый момент времени сразу несколько задач находятся в процессе выполнения. Например, для персонального компьютера характерно конкурентное выполнение таких задач, как копирование файлов, компиляция исходного кода, печать на принтере, отправка и получение писем, воспроизведение потокового аудио и видео, загрузка файлов из интернета и т. д.

Когда мы говорим, что две задачи выполняются **параллельно (in parallel)**, мы имеем в виду, что они выполняются одновременно в буквальном смысле. Таким образом, параллелизм — это подмножество конкурентности. Человеческий организм параллельно выполняет множество действий. Дыхание, кровообращение, пищеварение, мышление и ходьба — это параллельные процессы. Все органы чувств функционируют параллельно: зрение, слух, осязание, обоняние и вкус. Считается, что такой параллелизм возможен потому, что человеческий мозг содержит миллиарды «процессоров». Современные компьютеры с многоядерными процессорами могут выполнять несколько задач параллельно.

Конкурентность в C++

В C++ вы можете реализовать конкурентность средствами самого языка и стандартной библиотеки, используя несколько потоков выполнения — конкурентных процессов в программе. Согласно стандарту C++, **поток выполнения (thread of execution)**, для краткости называемый просто **потоком (thread)**,[1] — это одиночный

[1] Слово «thread» означает «нить», поэтому опытные программисты старой школы иногда называют потоки выполнения «нитями», сохраняя смысл английского термина. Но это слово в русском языке не прижилось: теперь «thread» — это «поток», а процессоры и программы называют не «многонитевыми», а многопоточными. — *Примеч. ред.*

поток управления (flow of control), содержащий первоначальный вызов любой конкретной функции верхнего уровня и рекурсивно включающий в себя все последующие вызовы функций, последовательно выполняемые этим потоком[1].

Программа может иметь несколько потоков выполнения, в каждом из которых есть свой стек вызовов функций и свой программный счетчик, что позволяет этому потоку выполняться, конкурируя с другими потоками. Все потоки программы могут совместно использовать ее общие ресурсы, такие как память и файлы. Эта функциональность называется **многопоточностью (multithreading)**.

Проблема однопоточных приложений, которая может приводить к замедленной реакции компьютера, заключается в том, что любая длительная операция должна завершиться до того, как смогут начаться какие-либо другие. Зато в многопоточном приложении потоки могут быть распределены между несколькими ядрами процессора, если эти ядра доступны. Это позволяет задачам выполняться параллельно, и тогда приложение может работать более эффективно.

Многопоточность часто повышает быстродействие приложений даже на компьютере с одноядерным процессором. Когда один поток не может продолжить работу (например, ожидает результата операции ввода-вывода), другой поток получает процессор в свое распоряжение..

Пример конкурентного программирования: просмотр видео из интернета

Мы обсудим различные сферы применения **конкурентного программирования (concurrent programming)**. Например, пользователи хотят начинать просмотр видео из интернета, не дожидаясь, пока закончится загрузка всего длинного файла. Для решения этой проблемы можно использовать два потока. Первый загружает видео маленькими фрагментами, по одному фрагменту за раз; мы будем называть такой поток **производителем (producer)**. Второй поток воспроизводит загруженные фрагменты; мы будем называть такой поток **потребителем (consumer)**. Эти действия могут выполняться конкурентно. Оба потока синхронизированы, чтобы избежать прерывистого воспроизведения. Их действия согласованы таким образом, чтобы потребитель не начинал работу до тех пор, пока в памяти не накопится столько фрагментов, сколько нужно для бесперебойного воспроизведения. Производитель и потребитель совместно используют данные. Мы покажем, как синхронизировать потоки, чтобы корректно выполнять подобные задачи. В случае потокового видео синхронизация потоков обеспечивает плавный просмотр, даже если в каждый момент времени в памяти находится только часть файла, содержащего фильм.

Потокобезопасность

Когда потоки совместно используют **изменяемые (mutable)**, то есть **модифицируемые (modifiable)**, данные, необходимо убедиться, что данные не будут повреждены. Если повреждение данных исключено, такой код называется **по-**

---

[1] Стандарт C++, «Multi-Threaded Executions and Data Races». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/intro.multithread#def:thread_of_execution.

токобезопасным (thread-safe). Потокобезопасность достигается несколькими способами, такими как[1]:

- Неизменяемые (константные) данные. Константный объект невозможно изменить, поэтому любое количество потоков может одновременно получать доступ к константному объекту.
- Взаимное исключение. Вы можете координировать доступ к общим изменяемым данным, разрешая обращаться к ним только одному потоку в каждый момент времени.
- Атомарные типы. Вы можете использовать низкоуровневые типы данных, которые автоматически обеспечивают атомарность (непрерываемость) выполняемых с ними операций. В каждый момент времени только один из потоков может получать доступ к данным, в том числе изменять их.
- Локальные данные потока. Если в многопоточном коде статическая или глобальная переменная объявлена с ключевым словом `thread_local` (C++11), каждый поток будет иметь собственную копию этой переменной. Переменные, объявленные как `thread_local`, не используются несколькими потоками совместно, поэтому не нарушают потокобезопасность кода[2,3].

Конкурентное программирование — это сложно

Создание многопоточных программ может быть непростым делом. Хотя человеческий разум способен выполнять все свои функции одновременно, даже человеку трудно переключаться от одной параллельной мысли к другой. Попробуйте провести эксперимент, чтобы понять, почему многопоточные программы могут быть сложными для написания и для чтения кода. Откройте на первых страницах три книги на совершенно разные темы и попробуйте читать их «в трех потоках». Прочитайте строку из первой книги, затем строку из второй, затем строку из третьей. Вернитесь к первой книге, прочитайте следующую строку, и т. д. После этого эксперимента вы оцените некоторые сложности многопоточности: вам пришлось переключаться между книгами, читать маленькими фрагментами, запоминать точку продолжения в каждой книге, придвигать к себе ту книгу, которую вы читаете, и отодвигать остальные (чтобы сохранять одинаковое расстояние от глаз до текста). И на фоне всего этого хаоса надо очень постараться, чтобы понять смысл всех книг!

C++11 и C++14: низкоуровневые инструменты конкурентности

До C++11 библиотеки многопоточности C++ были нестандартными расширениями, специфичными для конкретных платформ. В C++11 появилась

---

[1] «Thread Safety». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Thread_safety.

[2] Но потокобезопасность может быть нарушена, если в вашем коде будут доступные нескольким потокам указатели или ссылки на переменные, объявленные как thread_local.

[3] Пол Э. Маккенни (Paul E. McKenney) и Ж. Ф. Бастьен (J. F. Bastien). «Use Cases for Thread-Local Storage», 20 ноября 2014 г. Был доступен: 15 сентября 2023 г. https://wg21.link/n4324.

стандартизированная многопоточность. Для C++11 и C++14 Комитет по стандартам C++ определил только низкоуровневые примитивы. Затем они были использованы для создания более высокоуровневых средств, введенных в C++17 и C++20. Они также будут основой для реализации других высокоуровневых инструментов, которые появятся в C++23 и последующих версиях. C++11 стал первой версией C++, содержащей в стандартной библиотеке инструменты для написания многопоточных приложений, включая низкоуровневые примитивы синхронизации потоков, которые называются **мьютексом (mutex)** и **блокировкой (lock)**[1]. В C++14 были добавлены **разделяемые (shared)** мьютексы и блокировки[2].

C++17 и C++20: удобные высокоуровневые средства многопоточности

Высокоуровневые средства многопоточности, представленные в C++17 и C++20, значительно облегчают работу программистов. Это относится и к функциям, предложенным для C++23, которые мы кратко упомянем в разделе 17.15. Высокоуровневые инструменты очень важны, потому что они:

- упрощают многопоточное программирование;
- помогают находить конфликты потоков в таком программном обеспечении, где обычно нет возможности выяснить точные обстоятельства появления каждой ошибки;
- помогают избежать самых распространенных ошибок;
- упрощают сопровождение программ.

В C++17 были введены 69 параллельных алгоритмов стандартной библиотеки (упомянутых в разделах 14.8 и 14.9)[3]. В C++20 добавлены высокоуровневые средства синхронизации потоков: защелки, барьеры и семафоры (они еще не поддерживаются компилятором clang++ на момент написания книги), а также дополнительные параллельные алгоритмы и корутины (мы расскажем о корутинах в главе 18)[4]. Остальная часть этой главы познакомит вас с имеющимися в C++ средствами создания многопоточных приложений.

17.2. Параллельные алгоритмы стандартной библиотеки (C++17)

Вычислительная мощность компьютеров продолжает расти, но закон Мура[5] уже практически неприменим, поэтому теперь для повышения быстродействия про-

---

[1] «C++11». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/C++11.

[2] «C++14». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/C++14.

[3] «C++17». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/C++17.

[4] «C++20». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/C++20.

[5] «Moore's Law». https://en.wikipedia.org/wiki/Moore's law.

изводители компьютерного оборудования полагаются на многоядерные процессоры. В этой главе мы делаем акцент на высокоуровневых подходах к созданию многопоточных приложений. Для начала мы покажем вам введенные в C++17 параллельные перегрузки алгоритмов стандартной библиотеки (то есть перегруженные версии алгоритмов, рассчитанные на параллельную обработку). Они дают выигрыш при многопоточном выполнении, так как используют преимущества многоядерных архитектур и быстродействующей «векторной математики»[1, 2]. Векторные операции выполняют одну и ту же задачу над многими элементами данных одновременно, используя инструкции SIMD («single instruction, multiple data» — «одна инструкция, много данных»), которые есть во многих центральных и графических процессорах[3, 4].

17.2.1. Пример: сравнение быстродействия последовательного и параллельного алгоритмов сортировки

В разделе 6.12 мы сортировали алгоритмом `std::sort` массив `std::array` в одном потоке выполнения. Давайте сравним этот алгоритм с его перегруженной параллельной версией, чтобы убедиться, что она работает быстрее. Показанная на илл. 17.1 программа сортирует `100 000 000` случайно сгенерированных целых чисел, хранящихся в контейнерах `vector`. Мы используем функции из заголовка `<chrono>`, обрабатывающие значения времени, чтобы продемонстрировать ускорение параллельной сортировки по сравнению с последовательной сортировкой на одном и том же многоядерном компьютере.

Мы скомпилировали эту программу в Visual C++ и запустили ее на компьютере с 8-ядерным процессором Intel и 64-битной ОС Windows 10. Ваши результаты, вероятно, будут отличаться в зависимости от конфигурации компьютера, операционной системы и компилятора, а также от загруженности компьютера во время выполнения программы. Обратите внимание, что в этом примере работают алгоритмы сортировки, которым нужны итераторы произвольного доступа, а не диапазон с произвольным доступом. У алгоритмов библиотеки `std::ranges` C++20 еще нет параллельных версий, но они могут появиться в C++23[5].

---

[1] Билли О'Нил (Billy O'Neal), «Using C++17 Parallel Algorithms for Better Performance», 11 сентября 2018 г. Был доступен: 15 сентября 2023 г. https://devblogs.microsoft.com/cppblog/using-c17-parallel-algorithms-for-better-performance.

[2] Дитмар Куль (Dietmar Kuhl), «C++17 Parallel Algorithms», 28 октября 2017 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=Ve8cHE9LNfk.

[3] Стандарт C++, «General Utilities Library — Execution Policies — Unsequenced Execution Policy». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/execpol.unseq.

[4] «SIMD». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/SIMD.

[5] Барри Ревзин (Barry Revzin), Конор Хукстра (Conor Hoekstra) и Тим Сонг (Tim Song), «A Plan for C++23 Ranges», 14 октября 2020 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p2214r0.

```
// fig17_01.cpp
// Сравнение последовательной и параллельной версий алгоритма std::sort
#include <algorithm>
#include <chrono> // Для операций со значениями времени
#include <execution> // Для политик выполнения
#include <iostream>
#include <iterator>
#include <random>
#include <vector>

int main() {
   // Настраиваем генерацию случайных чисел
   std::random_device rd;
   std::default_random_engine engine{rd()};
   std::uniform_int_distribution ints{};

   std::cout << "Creating a vector v1 to hold 100 000 000 ints\n";
   std::vector<int> v1(100'000'000); // 100 000 000 элементов в векторе

   std::cout << "Filling vector v1 with random ints\n";
   std::generate(v1.begin(), v1.end(), [&](){return ints(engine);});

   // Копируем v1, получая одинаковые наборы данных для сравнения алгоритмов
   std::cout << "Copying v1 to vector v2 to create identical data sets\n";
   std::vector v2{v1};

   // Средства библиотеки <chrono> для измерения интервалов времени
   using std::chrono::steady_clock;
   using std::chrono::duration_cast;
   using std::chrono::milliseconds;

   // Сортируем v1 последовательным алгоритмом
   std::cout << "\nSorting 100 000 000 ints sequentially\n";
   auto start1{steady_clock::now()}; // Получаем текущее время
   std::sort(v1.begin(), v1.end()); // Сортируем
   auto end1{steady_clock::now()}; // Получаем текущее время

   // Вычисляем и отображаем длительность сортировки в миллисекундах
   auto time1{duration_cast<milliseconds>(end1 - start1)};
   std::cout << "Time: " << (time1.count() / 1000.0) << " seconds\n";

   // Сортируем v2 параллельным алгоритмом
   std::cout << "\nSorting the same 100 000 000 ints in parallel\n";
   auto start2{steady_clock::now()}; // Получаем текущее время
   std::sort(std::execution::par, v2.begin(), v2.end()); // Сортируем
   auto end2{steady_clock::now()}; // Получаем текущее время

   // Вычисляем и отображаем длительность сортировки в миллисекундах
   auto time2{duration_cast<milliseconds>(end2 - start2)};
   std::cout << "Time: " << (time2.count() / 1000.0) << " seconds\n";
}
```

Окончание ⇨

```
Creating a vector v1 to hold 100 000 000 ints
Filling vector v1 with random ints
Copying v1 to vector v2 to create identical data sets

Sorting 100 000 000 ints sequentially
Time: 8.296 seconds
Sorting the same 100 000 000 ints in parallel
Time: 1.227 seconds
```

Илл. 17.1 | Сравнение последовательной и параллельной версий алгоритма std::sort

Настройка генерации случайных чисел

В строках 13–15 задаются свойства случайных чисел, предназначенных для сортировки. Мы не указали диапазон случайных чисел, поэтому функция распределения `uniform_int_distribution` по умолчанию генерирует целые числа в диапазоне от `0` до `std::numeric_limits<int>::max()`, то есть до максимально возможного числа типа `int` в системе.

Создание массивов

В строке 18 создается вектор для хранения `100 000 000` чисел типа `int`, а в строке 21 вызывается функция `std::generate` для заполнения вектора случайными числами типа `int`. В строке 25 вектор копируется, чтобы мы могли сравнить последовательную и параллельную сортировку двух векторов с одинаковым содержимым.

Операции со значениями времени с помощью std::chrono

Для измерения длительности последовательной и параллельной сортировки мы используем средства стандартной библиотеки C++, которые находятся в заголовке `<chrono>`. Строки 28–30 показывают, что для этого нужны идентификаторы `steady_clock`, `duration_cast` и `milliseconds` из пространства имен `std::chrono`:

- С помощью объекта `steady_clock` мы получаем два значения времени: перед началом операции сортировки и после ее завершения. Объекты этого типа рекомендуются для операций хронометража[1]. Если вам нужно узнать текущее время суток, вы можете использовать объект `system_clock`, который возвращает значение времени, прошедшего с 00:00 1 января 1970 года[2].
- Шаблон функции `duration_cast` преобразует интервал времени в другие единицы измерения. В данном случае мы вычисляем разность между двумя моментами времени, затем вызываем `duration_cast` для преобразования результата в миллисекунды.

---

[1] «std::chrono::steady_clock». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/chrono/steady_clock.

[2] «std::chrono::system_clock». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/chrono/system_clock.

- Мы используем тип `milliseconds` из `std::chrono::duration` и функцию `duration_cast` для измерения суммарной длительности каждого вызова функции `sort`.

Последовательная сортировка

В строках 33–40 тестируется последовательная сортировка и измеряется ее длительность. В строках 34 и 36 переменным присваиваются два значения текущего времени: до и после вызова функции `sort` в строке 35. В строке 39 вычисляется разность между `end1` и `start1`, затем результат преобразуется в миллисекунды, которые возвращаются в виде объекта `std::chrono::duration`. В строке 40 вызывается функция `count` объекта `duration`, возвращающая длительность сортировки в миллисекундах, затем выполняется деление на `1000.0`, чтобы вывести результат в секундах.

Параллельная сортировка с политикой выполнения std::execution::par

В строках 43–50 тестируется параллельная сортировка и измеряется ее длительность. В строках 44 и 46 переменным присваиваются два значения текущего времени: до и после вызова функции `sort` в строке 45. Каждая перегруженная версия параллельного алгоритма требует в качестве первого параметра **политики выполнения (execution policy)**, указывающей, следует ли распараллеливать задачу, и если да, то каким способом. В строке 45 вызывается функция `sort` с политикой выполнения `std::execution::par` (заголовок `<execution>`), которая означает, что алгоритм должен выполнить задачу по частям на нескольких процессорных ядрах одновременно, если есть такая возможность. В разделе 17.2.3 рассматриваются четыре стандартные политики выполнения.

В строке 49 вычисляется разность между `end2` и `start2`, затем результат переводится в миллисекунды. В строке 50 это значение делится на `1000.0` и выводится результат в секундах. Выходные данные этой программы показывают, что на нашем компьютере параллельный алгоритм выполнил сортировку в `6.76` раза быстрее, чем последовательный. Очевиден прирост быстродействия при параллельной сортировке большого объема данных на нескольких ядрах.

Осторожно: параллельные алгоритмы не всегда быстрее последовательных

Нельзя утверждать, что параллельные алгоритмы всегда повышают быстродействие. Иногда параллельные алгоритмы работают медленнее, чем аналогичные последовательные алгоритмы[1]. Это особенно верно при обработке небольшого числа элементов и при использовании итераторов без произвольного доступа. В этих случаях затраты вычислительных ресурсов на распараллеливание могут перевесить преимущества параллельной обработки. Реализация стандартной библиотеки C++ от Microsoft по умолчанию переводит некоторые параллельные

[1] Лучан Раду Теодореску (Lucian Radu Teodorescu), «A Case Against Blind Use of C++ Parallel Algorithms», 4–7 февраля 2021 г. Был доступен: 15 сентября 2023 г. https://accu.org/journals/overload/29/161/teodorescu.

алгоритмы в режим последовательного выполнения, поскольку бенчмарки показали, что параллельные версии работают медленнее на тех видах компьютеров, на которые ориентирован Visual C++[1]. По этим причинам параллельные версии алгоритмов `copy`, `copy_n`, `fill`, `fill_n`, `move`, `reverse`, `reverse_copy`, `rotate`, `rotate_copy` и `swap_ranges` от Microsoft выполняются по умолчанию последовательно. Билли О'Нил из Microsoft рекомендует рассматривать возможность перехода на параллельные алгоритмы только в таких задачах, которые обрабатывают не менее 2000 элементов и требуют времени более $O(n)$ (например, в задачах сортировки)[2].

17.2.2. Когда нужны параллельные алгоритмы

Чтобы показать, что параллельное выполнение может увеличить время сортировки небольших наборов данных, мы запустили предыдущую программу (илл. 17.1), заполнив векторы случайными целыми числами от `100` до `100 000 000` с шагом `10`, и измерили время выполнения в наносекундах. Мы использовали Visual C++ на двух компьютерах с 64-битной ОС Windows 10:

- на нашем основном компьютере с 4-ядерным процессором;
- на более мощном компьютере с 8-ядерным процессором (мы обычно запускаем на нем ресурсоемкие задачи машинного обучения и глубокого обучения моделей искусственного интеллекта).

В таблице показаны сравнительные результаты. На каждом из двух компьютеров параллельная версия алгоритма начала немного опережать последовательную версию только при сортировке 10 000 случайных целых чисел. По мере увеличения количества чисел преимущество параллельного алгоритма стало более значительным.

Количество сортируемых элементов	Время последовательного выполнения на 4-ядерном процессоре, нс	Время параллельного выполнения на 4-ядерном процессоре, нс	Время последовательного выполнения на 8-ядерном процессоре, нс	Время параллельного выполнения на 8-ядерном процессоре, нс
100	3200	81 900	2800	63 400
1000	42 500	161 900	33 300	136 400
10 000	880 400	711 400	433 900	431 600
100 000	10 205 300	6 308 200	5 888 300	1 289 500
1 000 000	98 959 700	27 816 100	75 358 800	12 486 400
10 000 000	1 065 163 900	415 386 000	814 166 200	126 300 900
100 000 000	12 361 988 600	3 444 056 800	8 473 599 400	1 230 407 100

[1] Билли О'Нил (Billy O'Neal), «Using C++17 Parallel Algorithms for Better Performance».

[2] Билли О'Нил, «Using C++17 Parallel Algorithms for Better Performance».

17.2.3. Политики выполнения

На илл. 17.1 показан пример с политикой выполнения `std::execution::par`. Есть четыре стандартные политики выполнения[1]:

- `std::execution::seq` (C++17) указывает, что алгоритм должен выполняться в одном потоке. Это объект класса `std::execution::sequenced_policy`.
- `std::execution::par` (C++17) указывает, что алгоритм может быть распараллелен. Это объект класса `std::execution::parallel_policy`.
- `std::execution::par_unseq` (C++17) указывает, что алгоритм может быть распараллелен и векторизован. Это объект класса `std::execution::parallel_unsequenced_policy`. Напомним, что аппаратно реализованные векторные операции выполняют одну и ту же задачу над многими элементами данных одновременно.
- `std::execution::unseq` (C++20) указывает, что алгоритм может быть векторизован. Это объект класса `std::execution::unsequenced_policy`.

Кроме того, компиляторы могут предоставлять дополнительные политики выполнения[2].

Написанные на C++ программы используются на многих устройствах и операционных системах, часть из которых не поддерживают параллелизм. Поэтому политики выполнения являются лишь рекомендациями, то есть компиляторы могут игнорировать их или обрабатывать по-разному. Например, если компилятор ориентирован на аппаратное обеспечение, не поддерживающее векторизацию, он может по умолчанию установить политику `par` вместо `par_unseq`. В компиляторе Visual C++ «параллельная политика реализована так же, как параллельная векторизованная, поэтому не стоит ожидать лучшего быстродействия при использовании `par_unseq`»[3]. Протестируйте свой код и сравните скорость работы последовательных, параллельных и векторизованных алгоритмов, чтобы выяснить, когда имеет смысл использовать параллельные перегрузки алгоритмов.

17.2.4. Пример: сравнение быстродействия параллельных и векторных операций

На илл. 17.2 показана программа, сравнивающая быстродействие алгоритма `std::transform` при параллельном выполнении с политикой `std::execution::par` и при векторизованном выполнении с политикой `std::execution::unseq`:

- В строках 37–38 создаются векторы (контейнеры `vector`), содержащие 100 миллионов и 1 миллиард элементов соответственно, а в строках 41–44 они заполняются случайными числами типа `int`.

---

[1] «std::execution::seq, std::execution::par, std::execution::par_unseq, std::execution::unseq». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag.

[2] Стандарт C++, «20.18.1. Execution Policies». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/execpol#general.

[3] Билли О'Нил (Billy O'Neal), «Using C++17 Parallel Algorithms for Better Performance».

- В строках 49–58 вызывается сокращенный шаблон функции `timeTransform` (строки 13–28) для вычисления длительности операций `std::transform` над каждым контейнером с разными политиками выполнения. В строках 61–67 отображаются результаты расчета времени.
- Первым аргументом функции `timeTransform` является политика выполнения, которую мы передаем в `std::transform` (строка 21). В строках 21–22 вычисляется квадратный корень из каждого элемента контейнера, указанного во втором аргументе `timeTransform`, и результаты записываются в другой вектор.
- В строках 20 и 23 вызывается алгоритм `std::transform`, затем в строках 26–27 вычисляется и возвращается длительность в секундах.

Мы скомпилировали и запустили этот пример с помощью g++ 11.2 на MacBook Pro с процессором Intel. На этом компьютере вычисления с политикой `std::execution::unseq` выполняются примерно вдвое быстрее, чем с политикой `std::execution::par`. Как и в примере на илл. 17.1, ваши результаты, скорее всего, будут отличаться в зависимости от вашего компьютера, операционной системы и компилятора, а также от нагрузки на систему во время выполнения программы.

```
 1 // fig17_02.cpp
 2 // Преобразование элементов контейнеров с политиками выполнения par и unseq
 3 #include <algorithm>
 4 #include <chrono> // Для операций со значениями времени
 5 #include <cmath>
 6 #include <execution> // Для политик выполнения
 7 #include <fmt/format.h>
 8 #include <iostream>
 9 #include <random>
10 #include <vector>
11
12 // Измеряем и возвращаем длительность вызова std::transform в секундах
13 double timeTransform(auto policy, const std::vector<int>& v) {
14    // Средства библиотеки <chrono> для измерения интервалов времени
15    using std::chrono::steady_clock;
16    using std::chrono::duration_cast;
17    using std::chrono::milliseconds;
18
19    std::vector<double> result(v.size());
20    auto start{steady_clock::now()}; // Получаем текущее время
21    std::transform(policy, v.begin(), v.end(),
22       result.begin(), [](auto x) {return std::sqrt(x);});
23    auto end{steady_clock::now()}; // Получаем текущее время
24
25    // Вычисляем и отображаем длительность сортировки в секундах
26    auto time{duration_cast<milliseconds>(end - start)};
27    return time.count() / 1000.0;
28 }
29
30 int main() {
```

```
    // Настраиваем генерацию случайных чисел
    std::random_device rd;
    std::default_random_engine engine{rd()};
    std::uniform_int_distribution ints{0, 1000};

    std::cout << "Creating vectors\n";
    std::vector<int> v1(100'000'000);
    std::vector<int> v2(1'000'000'000);

    std::cout << "Filling vectors with random ints\n";
    std::generate(std::execution::par, v1.begin(), v1.end(),
       [&]() {return ints(engine);});
    std::generate(std::execution::par, v2.begin(), v2.end(),
       [&]() {return ints(engine);});

    std::cout << "\nCalculating square roots:\n";

    // Измеряем время обработки 100 000 000 элементов
    std::cout << fmt::format("{} elements with par\n", v1.size());
    double parTime1{timeTransform(std::execution::par, v1)};
    std::cout << fmt::format("{} elements with unseq\n", v1.size());
    double unseqTime1{timeTransform(std::execution::unseq, v1)};

    // Измеряем время обработки 1 000 000 000 элементов
    std::cout << fmt::format("{} elements with par\n", v2.size());
    double parTime2{timeTransform(std::execution::par, v2)};
    std::cout << fmt::format("{} elements with unseq\n", v2.size());
    double unseqTime2{timeTransform(std::execution::unseq, v2)};

    // Возвращаем таблицу с результатами измерений
    std::cout << "\nExecution times (in seconds):\n\n"
       << fmt::format("{:>13}{:>17}{:>21}\n", "# of elements",
          "par (parallel)", "unseq (vectorized)")
       << fmt::format("{:>13}{:>17.3f}{:>21.3f}\n",
          v1.size(), parTime1, unseqTime1)
       << fmt::format("{:>13}{:>17.3f}{:>21.3f}\n",
          v2.size(), parTime2, unseqTime2);
}
```

```
Creating vectors
Filling vectors with random ints
Calculating square roots:
100000000 elements with par
100000000 elements with unseq
1000000000 elements with par
1000000000 elements with unseq
Execution times (in seconds):
# of elements par (parallel) unseq (vectorized)
100000000 2.401 1.215
1000000000 22.969 11.787
```

Илл. 17.2 | Преобразование элементов контейнеров с политиками выполнения par и unseq

17.2.5. Дополнительные сведения о параллельных алгоритмах

В C++17 были добавлены не только параллельные версии 69 ранее существовавших алгоритмов, но и 7 новых параллельных алгоритмов:

- `for_each_n`[1] применяет функцию к первым `n` элементам диапазона.
- `exclusive_scan`[2] и `inclusive_scan`[3] — это параллельные версии алгоритма `partial_sum` (представленного в разделе 14.4.13). Отличие между ними заключается в том, что `exclusive_scan` не включает `n`-й элемент при вычислении суммы `n` элементов, а `inclusive_scan` включает. Как и `partial_sum`, оба параллельных алгоритма могут быть настроены на выполнение других операций вместо сложения.
- `transform_exclusive_scan`[4] и `transform_inclusive_scan`[5] аналогичны алгоритмам `exclusive_scan` и `inclusive_scan` соответственно, но применяют функцию преобразования к каждому элементу перед суммированием (или другой операцией).
- `reduce` получает одно значение из диапазона значений (например, вычисляет сумму элементов контейнера или находит в контейнере элемент с максимальным значением).
- `transform_reduce` выполняет преобразование каждого элемента в диапазоне или каждой пары элементов в паре диапазонов, а затем выполняет свертку результатов к одному значению.

Имена параллельных алгоритмов

Некоторые параллельные алгоритмы имеют названия, отличные от их последовательных эквивалентов. Например, алгоритм `reduce` является параллельным эквивалентом алгоритма `accumulate`[6].

В отличие от `accumulate`, алгоритм `reduce` не гарантирует порядок обработки элементов, что позволяет распараллелить `reduce` для повышения быстродействия[7].

---

[1] «std::for_each_n». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/algorithm/for_each_n.

[2] «std::exclusive_scan». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/algorithm/exclusive_scan.

[3] «std::inclusive_scan». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/algorithm/inclusive_scan.

[4] «std::transform_exclusive_scan». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/algorithm/transform_exclusive_scan.

[5] «std::transform_inclusive_scan». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/algorithm/transform_inclusive_scan.

[6] Дитмар Куль (Dietmar Kuhl), «C++17 Parallel Algorithms», 28 октября 2017 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=Ve8cHE9LNfk.

[7] Сай Бренд (Sy Brand), «std::accumulate vs. std::reduce», 15 мая 2018 г. Был доступен: 15 сентября 2023 г. https://blog.tartanllama.xyz/accumulate-vs-reduce.

Ограничения

Если в документации алгоритма не указано иное, то функции, лямбды или функторы, которые вы передаете алгоритму, не должны изменять никакие объекты, на которые они ссылаются прямо или косвенно через свои аргументы[1]. Кроме того, алгоритмы стандартной библиотеки могут копировать получаемые в аргументах функторы, поэтому функторы, передаваемые алгоритмам стандартной библиотеки, не должны хранить информацию о внутреннем состоянии, которая может быть неверной, если функторы будут скопированы[2].

17.3. Многопоточное программирование

Компьютер с многоядерным процессором может предоставить вашей программе несколько процессорных ядер, одновременно работающих над разными задачами программы и ускоряющих ее выполнение. Чтобы в полной мере использовать преимущества многоядерной архитектуры, вам надо писать многопоточные приложения. Когда задачи программы распределяются между несколькими потоками, многоядерная система может запускать эти потоки параллельно при наличии достаточного количества ядер.

Когда вы запускаете любую программу на современном компьютере, задачи вашей программы «конкурируют за внимание» процессора (процессорных ядер) с операционной системой, с другими приложениями и с другими процессами, которые для вас выполняет операционная система. Все виды задач система обычно выполняет «в фоновом режиме». Когда вы запускаете наши примеры, время их выполнения зависит от быстродействия процессора вашего компьютера, от количества ядер в нем, а также от всех остальных программ, запущенных на компьютере одновременно. Это похоже на автомобильную поездку в супермаркет. На ее длительность сильно влияют дорожные условия, погода, аварии на дорогах и другие факторы. Иногда вы доедете за 10 минут, но в другой раз вам может потребоваться гораздо больше времени, например, в час пик или в плохую погоду.

17.3.1. Состояния потока и жизненный цикл потока

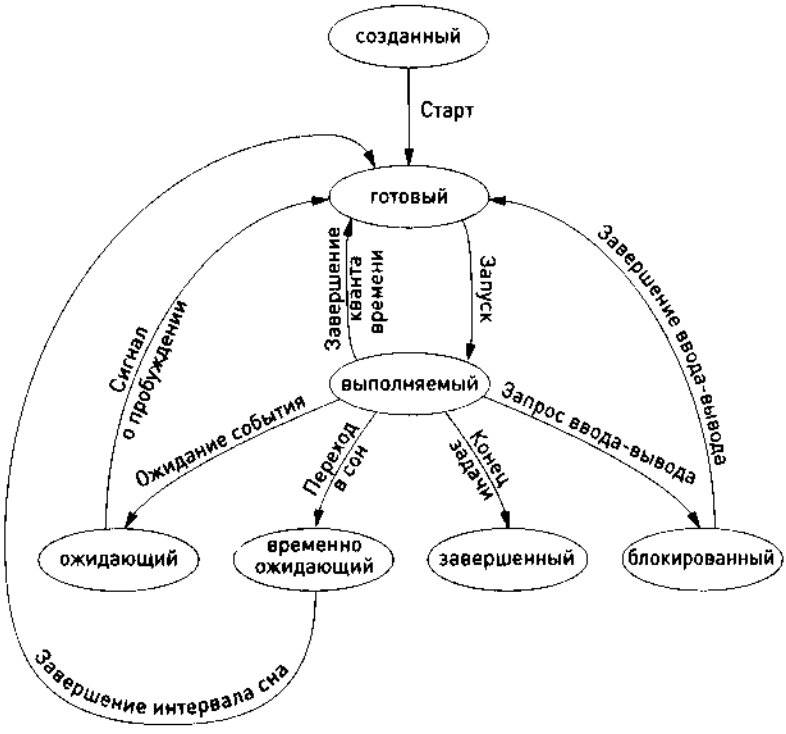
В любой момент времени поток находится в одном из нескольких возможных состояний. Вот диаграмма переключения состояний обычных потоков, которую мы позаимствовали из нашей книги об операционных системах[3]:

---

[1] Стандарт C++, «25.3.2. Requirements on User-Provided Function Objects». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/algorithms.parallel#user.

[2] Стандарт C++, «25.2. Algorithms Requirements». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/algorithms.requirements#10.

[3] Харви Дейтел (Harvey Deitel), Пол Дейтел (Paul Deitel) и Дэвид Чоффнес (David Choffnes), «Глава 4, Thread Concepts», «Operating Systems», 3-е изд., с. 153. Upper Saddle River, NJ: Prentice Hall, 2004 г.



Эта диаграмма должна дать вам представление о том, какие операции по переключению состояний выполняются «под капотом». Некоторые из этих терминов мы объясним позже. Примитивы библиотек параллелизма C++ скрывают большую часть этой сложности, значительно упрощая многопоточное программирование и делая его менее подверженным ошибкам.

Состояния born и ready

Новый поток начинает свой жизненный цикл в состоянии *born* («созданный») и остается в нем до тех пор, пока программа не запустит поток, что переводит его в состояние *ready* («готовый»). В C++ конструирование объекта потока с функцией в качестве аргумента (см. раздел 17.4) создает поток и немедленно его запускает, делая его готовым начать выполнение задачи, представленной этой функцией.

Состояние running

Готовый поток переключается в состояние *running* («выполняемый»), когда операционная система передает его процессору, — это называется **запуском потока (dispatching the thread)**. Операционная система периодически предоставляет каждому потоку очень короткие интервалы процессорного времени. Такой интервал называется **квантом времени**, или просто квантом (**quantum**, или **timeslice**).

Выбор длительности кванта — это один из важнейших вопросов при разработке операционных систем. Когда квант завершается, поток возвращается в состояние *ready*, а операционная система передает процессору другой поток. Переключения между состояниями *ready* и *running* осуществляются исключительно операционной системой. Она непрерывно занимается **планированием потоков (thread scheduling)** — так называется ее внутренний процесс принятия решений о том, какие потоки и когда надо передавать процессору. Решения по планированию должны приниматься очень аккуратно, чтобы обеспечить высокое быстродействие программ и избежать таких проблем, как неопределенно долгая отсрочка ожидающих потоков (см. раздел 17.3.2).

Состояние waiting

Иногда выполняемый поток переключается в состояние *waiting* («ожидающий»). Это необходимо, когда он должен дождаться выполнения задачи другим потоком. Ожидающий поток переключается обратно в состояние *ready*, когда другой поток уведомляет его о необходимости продолжить выполнение.

Состояние timed waiting

Выполняемый поток может переключиться в состояние *timed waiting* («временно ожидающий») на определенное время. Он переключается обратно в состояние *ready*, когда это время истекает или когда происходит событие, которого он ожидает. Как ожидающие, так и временно ожидающие потоки не могут использовать процессор, даже если он свободен.

Выполняемый поток может переключиться в состояние *timed waiting*, если он предоставляет опциональный интервал времени, пока ожидает выполнения задачи другим потоком. Такой поток возвращается в состояние *ready* при получении уведомления от другого потока или когда заданное время истекает.

Выполняемый поток может **засыпать (sleep)**, при этом он тоже переключается в состояние *timed waiting*. Спящий поток остается в этом состоянии в течение **интервала сна (sleep interval)**, после чего возвращается в состояние *ready*. Потоки спят, когда им временно не нужно выполнять работу. Например, в текстовом редакторе может быть поток, периодически сохраняющий документ в файле. Если бы поток не спал между операциями сохранения, то потребовался бы цикл, постоянно проверяющий, не пора ли уже сохранить документ. Это было бы неэффективно из-за расходования процессорного времени без полезной работы. Эффективнее задать потоку интервал сна, а затем переключить его в состояние *timed waiting*. Когда истечет интервал сна, поток переключится в состояние *ready*, сохранит документ и вернется в состояние *timed waiting*.

Состояние blocked

Выполняемый поток переключается в состояние *blocked* («блокированный»), когда пытается выполнить задачу, которая не может быть завершена немедленно. Он должен подождать, пока эта задача не завершится. Например, когда поток

выдает запрос ввода-вывода, операционная система блокирует выполнение потока до завершения операции ввода-вывода. В момент завершения этой операции блокированный поток переключается в состояние *ready*, чтобы возобновить выполнение. Блокированный поток не может использовать процессор, даже если он свободен.

Состояние terminated

Выполняемый поток переключается в состояние *terminated* («завершенный»), когда завершает свою задачу.

Планирование потоков

Квантование процессорного времени позволяет потокам совместно использовать процессор. Без квантования каждый поток выполнялся бы до конца, прежде чем другие потоки получили бы шанс на выполнение. Благодаря квантованию, даже если поток не закончил выполнение к моменту истечения кванта, операционная система забирает у этого потока процессор и отдает его следующему готовому потоку.

Планировщик потоков операционной системы решает, какой поток будет запущен следующим. В простейшей реализации планировщика потоков всем потокам предоставляется по одному кванту в порядке очереди. Этот процесс продолжается до тех пор, пока все потоки не завершат свою работу.

Планирование потоков зависит от платформы. Поведение многопоточной программы может отличаться в разных реализациях C++, на разных типах компьютеров и в разных операционных системах.

17.3.2. Тупики и неопределенно долгие отсрочки

Когда поток с более высоким приоритетом переключается в состояние готовности, операционная система, как правило, **вытесняет (preempts)** выполняемый поток с более низким приоритетом — это называется **вытесняющим планированием**. Постоянный приток потоков с высоким приоритетом может откладывать выполнение потоков с низким приоритетом, иногда на неопределенное время: это зависит от того, как устроена операционная система. Неопределенно долгая отсрочка выполнения потока приводит к его **зависанию**, или **голоданию (starvation)**. Чтобы потоки не зависали, операционные системы могут применять к ним **возрастное повышение приоритета**, или **старение (aging)**: когда поток долго ожидает в состоянии готовности, операционная система постепенно повышает его приоритет, чтобы гарантировать, что в конечном итоге он будет запущен.

Тупик (взаимная блокировка потоков)

Еще одна проблема, связанная с неопределенно долгой отсрочкой выполнения, — это **тупик (deadlock)**, который также называют взаимной блокировкой потоков. Поток находится в тупике, если он ожидает события, которое никогда не произой-

дет[1,2]. В многопоточных системах одной из важнейших задач является разделение ресурсов. Когда ресурсы разделяются между несколькими потоками, каждый из которых монопольно контролирует выделенные ему ресурсы, могут возникать тупиковые ситуации: некоторые потоки никогда не смогут завершить выполнение. Представьте, что ожидающий поток (назовем его `thread1`) не может продолжить выполнение, потому что находится в ожидании (прямо или косвенно) другого потока (назовем его `thread2`), а `thread2` не может продолжить выполнение, потому что ожидает (прямо или косвенно) `thread1`. Два потока ждут друг друга, поэтому события, которые позволили бы им продолжить выполнение, никогда не произойдут. Результатом может быть замедление работы и даже полный крах системы.

Четыре необходимых условия для возникновения тупика

Кофман, Элфик и Шошани[3] доказали, что для возникновения тупика необходимы следующие четыре условия:

1. Ресурс может быть получен для исключительного использования только одним потоком одновременно (условие взаимного исключения).
2. Поток, получивший эксклюзивный ресурс, может его удерживать, пока ожидает получения других ресурсов (условие ожидания, также называемое условием «hold-and-wait»).
3. После того как поток получил ресурс, система не может вернуть его из-под контроля потока, пока поток не закончит использование ресурса (условие «no-preemption»).
4. Несколько потоков (два или более) заблокированы в «замкнутой цепи», в которой каждый поток ожидает один или несколько ресурсов, занятых следующим потоком в цепи (условие кругового ожидания).

Поскольку это необходимые условия, все они должны быть в силе, чтобы в системе возник тупик. Запрет любого из этих четырех условий предотвращает возникновение тупиков. В совокупности все четыре условия являются необходимыми и достаточными для существования тупика (если все они выполняются, то система находится в тупике).

Неопределенно долгая отсрочка

Неопределенно долгая отсрочка возникает, когда поток, не находящийся в тупике, ждет события, которое может никогда не произойти (или произойдет непредсказуемо далеко в будущем) из-за неточностей политики планирования ресурсов в операционной системе. Гарантированное предотвращение тупиков

---

[1] С. С. Ислур (S. S. Isloor) и Т. А. Марсленд (T. A. Marsland), «The Deadlock Problem: An Overview», «Computer», том 13, № 9, сентябрь 1980 г., с. 58–78.

[2] Д. Зобель (D. Zobel), «The Deadlock Problem: A Classifying Bibliography», «Operating Systems Review», том 17, № 4, октябрь 1983 г., с. 6–16.

[3] Э. Г. Кофман-младший (E. G. Coffman, Jr.), М. Дж. Элфик (M. J. Elphick) и А. Шошани (A. Shoshani), «System Deadlocks», «Computing Surveys», том 3, № 2, июнь 1971 г., с. 69.

и неопределенно долгих отсрочек в операционной системе, как правило, может быть достигнуто только дорогой ценой. В критически важных системах эта цена должна быть заплачена независимо от того, насколько она высока, потому что развитие тупиковой ситуации или неопределенно долгой отсрочки может привести к катастрофе, особенно если это поставит под угрозу человеческую жизнь.

Предотвращение тупиковых ситуаций

Существуют различные методы предотвращения тупиковых ситуаций. Дж. Хавендер заметил, что тупик не может возникнуть, если система предотвращает любое из четырех необходимых условий, и предложил несколько стратегий предотвращения тупиков[1].

Наиболее практичный подход заключается в том, чтобы каждый поток запрашивал сразу все необходимые ресурсы и не продолжал работу до тех пор, пока все они не будут предоставлены. Если поток сначала запрашивает и получает все необходимые ему ресурсы, потом работает до завершения, а в итоге освобождает ресурсы, то «замкнутый круг» ожидания невозможен и поток не окажется в тупике. Если поток не может получить сразу все свои ресурсы, он должен отменить запрос и повторить попытку позже, позволяя другим потокам продолжить работу.

Запрос всех необходимых ресурсов сразу — это не идеальное решение, так как иногда приводит к неопределенно долгим отсрочкам и к плохому использованию системных ресурсов. Предположим, что поток выполняет 10-минутную задачу, требующую пяти ресурсов:

- один из них нужен на весь период выполнения задачи;
- остальные четыре нужны только в последнюю минуту выполнения задачи.

В этом случае вы получите низкую эффективность использования тех четырех ресурсов, которые не нужны до последней минуты.

Опасности, связанные с ожиданием

Тупики и неопределенно долгие отсрочки включают в себя те или иные формы ожидания. В конкурентном программировании эти сценарии ожидания часто развиваются незаметно, поэтому их нелегко обнаружить, особенно по мере роста числа активных конкурентных задач. Когда на карту поставлена жизнь людей, последствия могут быть катастрофическими. Программируя конкурентные задачи, вы должны развивать в себе здоровое чувство осторожности. Сможете ли вы создавать корректно функционирующие многопоточные системы? Да, это возможно. Легко ли это будет сделать? Не всегда. Важнейшим ключом к созданию надежных многопоточных систем, критически важных для жизнеобеспечения и бизнеса, является тенденция разработки и применения высокоуровневых при-

[1] Дж. В. Хавендер (J. W. Havender), «Avoiding Deadlock in Multitasking Systems», «IBM Systems Journal», том 7, № 2, 1968 г., с. 74–84.

митивов многопоточности. Этому высокоуровневому подходу посвящена значительная часть главы 17, но сначала мы рассмотрим несколько низкоуровневых «строительных блоков».

17.4. Запуск задач с помощью std::jthread

Стандарт C++ предоставляет два класса для запуска конкурентных задач в приложении: `std::thread` (C++11) и `std::jthread` (C++20)[1]. Оба класса определены в заголовке `<thread>`, который также включает служебные функции, такие как `get_id`, `sleep_for` и `sleep_until`. В этой главе мы используем исключительно класс `std::jthread`[2,3], поскольку в нем исправлен ряд проблем, имеющихся в `std::thread`.

Если вы хотите выделить в отдельный поток задачу для ее выполнения конкурентно с другими задачами:

- создайте функцию (или лямбду, или функциональный объект), которая будет выполнять задачу;
- затем инициализируйте объект `std::jthread` этой функцией (или лямбдой, или функциональным объектом), чтобы запустить ее в отдельном потоке.

Возвращаемое значение задачи игнорируется. Как вы скоро увидите, для передачи данных между потоками используются другие механизмы.

Исключения в объектах jthread

Если задача в объекте `std::jthread` завершается исключением, программа завершается вызовом функции `std::terminate`.

17.4.1. Определение задачи для выполнения в потоке

Этот пример кода состоит из заголовка `printtask.h` (илл. 17.3) и приложения с функцией `main` (илл. 17.4). Мы используем функцию `id` (илл. 17.3, строки 11–15) в инструкциях вывода, чтобы показать, какой поток выполняется в данный момент времени. Каждый поток имеет уникальный идентификатор (ID), включая поток `main`, потоки, созданные с помощью `std::jthread` или `std::thread`, а также потоки, созданные для вас библиотечными функциями. В строке 13 выражение

```
std::this_thread::get_id()
```

---

[1] Мы также продемонстрируем средства более высокого уровня, которые позволят вам не создавать потоки в явном виде.

[2] Класс std::jthread поддерживается в clang++ версии 13 и выше. Чтобы запустить использующие его примеры кода в более ранних версиях clang++, замените std::jthread на std::thread и присоедините (join) потоки, как показано в разделе 17.4.3.

[3] В Linux добавьте флаг компилятора -pthread к командам компиляции, чтобы использовать класс std::jthread.

вызывает функцию `get_id` из пространства имен `std::this_thread`, чтобы получить уникальный ID выполняющегося в данный момент потока, возвращаемый в виде объекта `std::thread::id`. Мы используем перегруженный оператор этого объекта `operator<<` для преобразования ID в строку `std::string`.

```
1 // Fig. 17.3: printtask.h
2 // Функция printTask определяет задачу для выполнения в отдельном потоке
3 #include <chrono>
4 #include <fmt/format.h>
5 #include <iostream>
6 #include <sstream>
7 #include <string>
8 #include <thread>
9
10 // Получаем ID текущего потока в виде строки
11 std::string id() {
12    std::ostringstream out;
13    out << std::this_thread::get_id();
14    return out.str();
15 }
16
17 // Задача для выполнения в отдельном потоке
18 void printTask(const std::string& name,
19    std::chrono::milliseconds sleepTime) {
20
21    // Средства библиотеки <chrono> для измерения интервалов времени
22    using std::chrono::steady_clock;
23    using std::chrono::duration_cast;
24    using std::chrono::milliseconds;
25
26    std::cout << fmt::format("{} (ID {}) going to sleep for {} ms\n",
27       name, id(), sleepTime.count());
28
29    auto startTime{steady_clock::now()}; // Получаем текущее время
30
31    // Переводим поток в сон на sleepTime миллисекунд
32    std::this_thread::sleep_for(sleepTime);
33
34    auto endTime{steady_clock::now()}; // Получаем текущее время
35    auto time{duration_cast<milliseconds>(endTime - startTime)};
36    auto difference{duration_cast<milliseconds>(time - sleepTime)};
37    std::cout << fmt::format("{} (ID {}) awakens after {} ms ({} + {})\n",
38       name, id(), time.count(), sleepTime.count(), difference.count());
39 }
```

Илл. 17.3 | Функция printTask определяет задачу для выполнения в отдельном потоке

Функция `printTask` (строки 18–39) реализует задачу, которую мы хотим выполнить. Параметры представляют собой:

- `name` — имя, которое мы используем для идентификации каждой задачи в выводе программы;
- `sleepTime` — интервал сна в миллисекундах, который мы используем, чтобы заставить выполняющийся поток освободить процессор как минимум на это время.

Когда поток вызывает функцию `printTask`:

- В строках 26–27 выводится сообщение с указанием имени выполняемой в данный момент задачи, уникального идентификатора потока и времени сна в миллисекундах[1].
- В строке 29 измеряется текущее время перед засыпанием потока.
- В строке 32 вызывается функция `sleep_for` из пространства имен `std::this_thread`, чтобы перевести поток в сон как минимум на указанный период времени. В этот момент поток освобождает процессор, что, возможно, позволяет другому потоку выполнить свою работу. В наших примерах потоки часто засыпают, чтобы имитировать выполнение работы. Мы также используем случайные интервалы сна, чтобы подчеркнуть, что невозможно предсказать, когда каждый поток получит процессорное время для выполнения своей задачи. Пространство имен `std::this_thread` также предоставляет функцию `sleep_until`, которая переводит поток в сон до определенного времени.
- Когда время сна потока истекает, поток снова переходит в состояние *ready*, но не обязательно сразу начинает выполняться. Затем, когда операционная система предоставляет потоку процессор, в строке 34 измеряется время, в строке 35 вычисляется суммарное время, в течение которого поток не выполнялся, а в строке 36 вычисляется разность между полным временем выполнения и временем сна. В строках 37–38 выводятся полученные значения времени.
- Когда `printTask` завершается, его `jthread` переходит в состояние *terminated*.

17.4.2. Выполнение задачи в потоке jthread

Показанная на илл. 17.4 программа запускает два конкурентных потока, выполняющих функцию `printTask` (илл. 17.3), и показывает два примера выходных данных. В строках 14–16 задается генерация случайных чисел для выбора случайного интервала сна до 5000 миллисекунд. В строке 18 создается вектор для хранения объектов `std::jthread`. Мы используем его для того, чтобы функция `main` могла ждать, пока потоки выполняют свои задачи перед завершением программы. Подробнее об этом мы расскажем чуть позже.

---

[1] Когда несколько потоков выводят данные на экран через std::cout, выводимые разными потоками фрагменты данных могут чередоваться, поэтому текст будет выглядеть поврежденным. Чтобы избежать этого, надо синхронизировать потоки (раздел 17.6).

```
// Fig. 17.4: printtask.cpp
// Конкурентное выполнение задач с помощью std::jthreads
#include <chrono>
#include <fmt/format.h>
#include <iostream>
#include <random>
#include <string>
#include <thread>
#include <vector>
#include "printtask.h"

int main() {
   // Инициализируем генератор случайных чисел
   std::random_device rd;
   std::default_random_engine engine{rd()};
   std::uniform_int_distribution ints{0, 5000};

   std::vector<std::jthread> threads; // Для хранения объектов jthread

   std::cout << "STARTING JTHREADS\n";

   // Запускаем два jthread
   for (int i{1}; i < 3; ++i) {
      std::chrono::milliseconds sleepTime{ints(engine)};
      std::string name{fmt::format("Tasks {}", i)};

      // Создаем jthread, который вызывает printTask, передавая в аргументах
      // name и sleepTime; сохраняем jthread, чтобы он не был уничтожен,
      // пока вектор не выйдет из области видимости в конце функции main;
      // деструктор каждого jthread автоматически присоединяет этот jthread
      threads.push_back(std::jthread{printTask, name, sleepTime});
   }

   std::cout << "\nJTHREADS STARTED\n";
   std::cout << "\nMAIN ENDS\n";
}
```

```
STARTING JTHREADS
JTHREADS STARTED
MAIN ENDS
Task 2 (ID 15704) going to sleep for 4547 ms
Task 1 (ID 15624) going to sleep for 3648 ms
Task 1 (ID 15624) awakens after 3651 ms (3648 + 3)
Task 2 (ID 15704) awakens after 4555 ms (4547 + 8)
```

```
STARTING JTHREADS
JTHREADS STARTED
MAIN ENDS
Task 1 (ID 9368) going to sleep for 441 ms
Task 2 (ID 16876) going to sleep for 2614 ms
Task 1 (ID 9368) awakens after 449 ms (441 + 8)
Task 2 (ID 16876) awakens after 2618 ms (2614 + 4)
```

Илл. 17.4 | Одновременное выполнение задач с помощью std::jthread

В строках 23–32 создаются два объекта `std::jthread`:

- В строке 24 выбирается случайный интервал сна.
- В строке 25 создается объект `string`, необходимый для идентификации задачи в выходных данных.
- В строке 31 создаются и добавляются в вектор объекты `jthread`. Конструктор `jthread` получает функцию для выполнения (`printTask`) и аргументы, которые должны быть переданы этой функции (`name` и `sleepTime`). Эта инструкция создает `jthread` как временный объект, поэтому функция `push_back` класса `vector`, получающая ссылку *rvalue*, перемещает объект в новый элемент вектора.

Конструирование `jthread` с функцией для выполнения запускает `jthread`, чтобы операционная система могла запланировать его выполнение на одном из ядер процессора. В строке 34 выводится сообщение о том, что потоки были запущены, а в строке 35 — о том, что функция `main` завершилась.

Ожидание завершения запланированных задач

После планирования выполнения задач обычно надо дождаться их завершения, например, для того, чтобы использовать их результаты. Вы сообщаете функции `main`, что она должна ждать завершения задачи потока `jthread`, **присоединяя (joining)** этот поток вызовом его функции `join`. Это можно сделать явно или, как мы делаем в этой программе, неявно через деструктор `jthread` — в этом заключается одно из преимуществ `jthread` по сравнению с `std::thread` из версии C++11 (см. раздел 17.4.3). Как вы знаете, когда функция `main` завершается, ее локальные переменные выходят из области видимости и вызываются их деструкторы. Когда выполняется деструктор объекта `vector`, выполняются деструкторы каждого элемента `jthread`, вызывая функцию `join` этого `jthread`. Когда все присоединенные потоки `jthread` завершат выполнение, программа может завершиться.

Основной поток

Код функции `main` выполняется в **основном потоке (main thread)**. Функция `printTask` выполняется, когда операционная система запускает соответствующий `jthread`, через некоторое время после его перехода в состояние готовности.

Выходные данные

Примеры вывода показывают имя каждой задачи и интервал сна в тех случаях, когда поток засыпает. Поток с наименьшим интервалом сна обычно пробуждается первым, но это не гарантируется. Когда каждый из потоков продолжает работу, он отображает имя своей задачи, уникальный идентификатор (ID) потока и значение времени, а затем завершается. Первый пример выходных данных показывает, что задачи переводятся в сон не в том порядке, в каком мы создали их потоки `jthread`. Мы не можем предсказать тот порядок, в каком задачи начнут выполняться, даже если знаем порядок их создания и запуска. Это одна из проблем многопоточного программирования.

17.4.3. Преимущества класса jthread перед классом thread

В руководстве C++ Core Guidelines говорится, что `std::jthread` (C++20) предпочтительнее, чем `std::thread`[1]. У класса `thread` есть ряд проблем. Как мы показали в предыдущем примере, `jthread` автоматически **присоединяет (join)** все потоки, чтобы гарантировать, что они завершат выполнение до завершения программы. Название `jthread` — это сокращение от «joining thread» («присоединенный поток»). С другой стороны, если вы не присоедините поток класса `thread` или отсоедините его до уничтожения, его деструктор вызовет функцию `std::terminate`, которая немедленно завершит программу[2]. Чтобы предотвратить это, вы должны явно присоединять каждый поток класса `thread`. Если бы в предыдущей программе использовались потоки класса `thread`, то перед закрывающей скобкой функции `main` надо было бы добавить примерно такой цикл:

```
for (auto& t : threads) {
   t.join();
}
```

Впрочем, даже с таким циклом сохраняется риск другой ошибки. Если функция, запустившая потоки класса `thread`, завершится до присоединения всех этих потоков из-за неперехваченного исключения, потоки будут уничтожены, а первый выполненный деструктор потока вызовет функцию `std::terminate`.

Вы обязательно должны присоединять каждый поток, поэтому `jthread` устраняет названную проблему, вызывая свою функцию `join` в своем деструкторе[3]. Конечно, деструктор выполняется каждый раз, когда `jthread` выходит из области видимости:

- в конце блока, создавшего `jthread`; или
- когда блок завершается из-за исключения.

Класс `jthread` также устраняет другие проблемы, имеющиеся в классе `thread`. В частности, `jthread` поддерживает кооперативное прерывание (раздел 17.9), корректно поддерживает семантику перемещения и является типом RAII (раздел 11.5), правильно освобождающим использованные им ресурсы[4].

---

[1] C++ Core Guidelines, «CP.25: Prefer gsl::joining_thread over std::thread». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconc-joining_thread.

[2] Отсоединение потока отделяет его от объекта thread или jthread, но операционная система продолжает выполнять этот поток. В C++ Core Guidelines рекомендуется избегать отсоединения, так как оно «затрудняет мониторинг отсоединенного потока и взаимодействие с ним». Подробнее: «CP.26: Don't detach() a Thread». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconc-detached_thread.

[3] Николай Хосуттис (Nicolai Josuttis), «Why and How We Fixed std::thread by std::jthread», 29 июля 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=elFil2VhlH8.

[4] Николай Хосуттис, «C++20: The Complete Guide», глава 15, «std::jthread and Stop Tokens», 21 ноября 2021 г. http://cppstd20.com.

17.5. Отношения между производителем и потребителем: первый эксперимент

Как вы скоро увидите, когда конкурентные потоки обращаются к общим для них неконстантным данным и хотя бы один из этих потоков изменяет эти данные, результаты могут быть непредсказуемыми:

- Если поток A вносит изменение в общий объект, а поток B тоже вносит изменение в этот объект, нельзя предсказать, какое из этих двух изменений вступит в силу.
- Аналогично, если поток A вносит изменение в общий объект, а поток B выполняет чтение этого объекта, нельзя предсказать, прочитает поток B старое значение или новое.

В таких случаях поведению программы нельзя доверять. Иногда результаты будут правильными, а иногда ошибочными. При этом вы не увидите никаких признаков того, что общий изменяемый объект был обработан некорректно. Хуже всего, что при некоторых запусках такой программы она может работать правильно, но при очередном запуске ее поведение станет непредсказуемым.

В этом и следующем разделах мы приведем два примера. Первый пример демонстрирует проблемы с доступом конкурентных потоков к общим изменяемым данным. Во втором примере вы увидите, как эти проблемы решаются. Оба примера демонстрируют отношения между **производителем (producer)** и **потребителем (consumer)**, в которых:

- поток-производитель генерирует данные и сохраняет их в объекте, общем для обоих потоков;
- поток-потребитель читает данные из этого объекта.

Пример отношений между производителем и потребителем

Очередь данных, отправляемых с компьютера на принтер, — распространенный пример отношений между производителем и потребителем. Принтер — это эксклюзивный ресурс. Он может быть недоступен в тот момент, когда вы хотите напечатать что-либо из приложения (из производителя), но вы все равно можете «завершить» задачу печати. Отправленные на принтер данные временно хранятся в буфере, который называется **спулером печати (print spooler)**, до тех пор, пока принтер не станет доступен. Соответственно, когда принтер (потребитель) доступен, он не ждет, пока текущий пользователь захочет что-то напечатать. Данные из спулера могут быть напечатаны, когда принтер становится доступным.

Синхронизация

В многопоточных отношениях между производителем и потребителем поток-производитель генерирует данные, помещая их в общий объект, который является буфером. Поток-потребитель читает данные из этого буфера. Такие отношения

требуют синхронизации, чтобы данные производились и потреблялись корректно. Все операции над общими изменяемыми данными, к которым обращаются конкурентные потоки, должны быть защищены блокировкой, предотвращающей повреждение данных, как мы покажем в разделе 17.6.

Зависимость от состояния буфера

Операции над общим буфером также зависят от его состояния. Операции должны выполняться только в том случае, если буфер находится в правильном состоянии:

- Если буфер находится в незаполненном состоянии, производитель может производить.
- Если буфер находится в непустом состоянии, потребитель может потреблять.

Аналогично:

- Если буфер находится в заполненном состоянии, когда производитель хочет записать новое значение, он должен подождать, пока в буфере освободится место.
- Если буфер находится в пустом состоянии, когда потребитель хочет прочитать значение, он должен подождать, пока новые данные станут доступны.

Логические ошибки из-за отсутствия синхронизации

Проиллюстрируем опасность одновременной работы потоков с изменяемыми данными без надлежащей синхронизации (илл. 17.5–17.6). Поток-производитель записывает числа от `1` до `10` в общий буфер: в данном случае в переменную типа `int` с именем `m_buffer` (илл. 17.5, строка 23). Поток-потребитель читает эти данные из общего буфера и выводит их на экран. Все это может показаться правильным, но на самом деле в выходных данных программы обнаружатся ошибки, возникающие из-за того, что ничем не ограниченный производитель произвольно записывает значения в общий буфер (производит), а ничем не ограниченный потребитель произвольно читает эти значения из общего буфера (потребляет).

Каждое значение, которое поток-производитель записывает в общий буфер, должно быть использовано потоком-потребителем ровно один раз. Потоки в этом примере не синхронизированы, поэтому:

- данные могут быть потеряны или искажены, если производитель запишет новые данные в общий буфер до того, как потребитель прочитает предыдущие данные;
- данные могут быть ошибочно продублированы, если потребитель повторно прочитает их до того, как производитель запишет следующее значение.

Чтобы продемонстрировать возможность этих ошибок, поток-потребитель в следующем примере суммирует все прочитанные им числа. Поток-производитель производит числа от 1 до 10. Если потребитель правильно прочитает каждое из них один и только один раз, то сумма этих чисел будет равна 55. Как вы увидите,

из-за отсутствия синхронизации между производителем и потребителем в результате могут быть получены неправильные суммы чисел (отличные от 55). Интересно, что ошибки в несинхронизированной паре, состоящей из производителя и потребителя, иногда могут приводить к правильной сумме 55. Но мы, конечно, хотим обеспечить корректную работу такой пары. Мы покажем, как это сделать, в разделе 17.6.

Класс UnsynchronizedBuffer

В программе, показанной на илл. 17.6, поток-производитель и поток-потребитель совместно работают с объектом класса `UnsynchronizedBuffer` (илл. 17.5). `UnsynchronizedBuffer` содержит один элемент данных `int` (строка 22). Поток-производитель вызывает функцию `put` этого класса (строки 11–14), чтобы записать значение в элемент `int`, а поток-потребитель вызывает функцию `get` этого класса (строки 17–20), чтобы прочитать значение из элемента `int`.

```
1 // Fig. 17.5: UnsynchronizedBuffer.h
2 // Ошибки в несинхронизированном буфере класса UnsynchronizedBuffer,
3 // к которому обращаются поток-производитель и поток-потребитель
4 #pragma once
5 #include <fmt/format.h>
6 #include <iostream>
7 #include <string>
8
9 class UnsynchronizedBuffer {
10 public:
11    // Записываем значение в буфер
12    void put(int value) {
13       std::cout << fmt::format("Producer writes\t{:2d}", value);
14       m_buffer = value;
15    }
16
17    // Читаем значение из буфера
18    int get() const {
19       std::cout << fmt::format("Consumer reads\t{:2d}", m_buffer);
20       return m_buffer;
21    }
22 private:
23    int m_buffer{-1}; // Буфер, общий для производителя и потребителя
24 };
```

Илл. 17.5 | Ошибки в несинхронизированном буфере класса UnsynchronizedBuffer, к которому обращаются поток-производитель и поток-потребитель (этот класс не является потокобезопасным)

Этот буфер не защищен синхронизацией

Класс `UnsynchronizedBuffer` не синхронизирует доступ к своим данным, поэтому не является потокобезопасным. В строке 23 элемент `m_buffer` класса `UnsynchronizedBuffer` инициализируется значением `-1`. Мы используем это

значение, чтобы показать случай, когда поток-потребитель пытается прочитать значение до того, как поток-производитель поместит его в `m_buffer`. Функция `put` просто присваивает свой аргумент переменной `m_buffer` (строка 14), а функция `get` просто возвращает значение переменной `m_buffer` (строка 20).

Программа с функцией main

В строке 12 (илл. 17.6) создается объект `buffer` класса `UnsynchronizedBuffer`, в котором хранятся данные, общие для двух конкурентных потоков: производителя и потребителя. Поток-производитель вызывает лямбду `produce` (строки 15–36), а поток потребителя — лямбду `consume` (строки 39–60). Скоро мы рассмотрим обе лямбды более подробно. Список захвата каждой лямбды захватывает `buffer` по ссылке, поэтому конкурентные потоки, выполняющие эти лямбды, используют этот объект совместно. В строках 62–63 на экран выводятся заголовки столбцов для выходных данных программы. В строках 65–66 создаются два потока `jthread`, производитель выполняет лямбду `produce`, а потребитель выполняет лямбду `consume`. Каждый из них выходит из области видимости в конце функции `main`, которая присоединяет потоки. Мы добавили к выходным данным наши комментарии, выделив их курсивом.

```
 1 // Fig. 17.6: SharedBufferTest.cpp
 2 // Конкурентные потоки jthread используют несинхронизированный буфер
 3 #include <chrono>
 4 #include <fmt/format.h>
 5 #include <iostream>
 6 #include <random>
 7 #include <thread>
 8 #include "UnsynchronizedBuffer.h"
 9
10 int main() {
11    // Создаем буфер UnsynchronizedBuffer для хранения целых чисел
12    UnsynchronizedBuffer buffer;
13
14    // Лямбда-выражение производит числа от 1 до 10 и суммирует их
15    auto produce{
16       [&buffer]() {
17          // Инициализируем генератор случайных чисел
18          std::random_device rd;
19          std::default_random_engine engine{rd()};
20          std::uniform_int_distribution ints{0, 3000};
21
22          int sum{0};
23
24          for (int count{1}; count <= 10; ++count) {
25             // Получаем случайный интервал сна, затем переводим поток в сон
26             std::chrono::milliseconds sleepTime{ints(engine)};
27             std::this_thread::sleep_for(sleepTime);
28
29             buffer.put(count); // Записываем число в буфер
30             sum += count; // Прибавляем count к sum
```

```
            std::cout << fmt::format("\t{:2d}\n", sum);
         }

         std::cout << "Producer done producing\nTerminating Producer\n";
      }
   };

   // Лямбда-выражение потребляет числа от 1 до 10 и суммирует их
   auto consume{
      [&buffer]() {
         // Инициализируем генератор случайных чисел
         std::random_device rd;
         std::default_random_engine engine{rd()};
         std::uniform_int_distribution ints{0, 3000};

         int sum{0};

         for (int count{1}; count <= 10; ++count) {
            // Получаем случайный интервал сна, затем переводим поток в сон
            std::chrono::milliseconds sleepTime{ints(engine)};
            std::this_thread::sleep_for(sleepTime);

            sum += buffer.get(); // Читаем число из буфера, прибавляем к sum
            std::cout << fmt::format("\t\t\t{:2d}\n", sum);
         }

         std::cout << fmt::format("\n{} {}\n{}\n",
            "Consumer read values totaling", sum, "Terminating Consumer");
      }
   };

   std::cout << "Action\t\tValue\tSum of Produced\tSum of Consumed\n";
   std::cout << "------\t\t-----\t---------------\t---------------\n";

   std::jthread producer{produce}; // Запускаем jthread-производитель
   std::jthread consumer{consume}; // Запускаем jthread-потребитель
}
```

```
Action          Value   Sum of Produced  Sum of Consumed
------          -----   ---------------  ---------------
Producer writes  1       1
Producer writes  2       3                               — число 1 потеряно
Consumer reads   2                        2
Consumer reads   2                        4              — число 2 прочитано повторно
Producer writes  3       6
Consumer reads   3                        7
Producer writes  4      10
Producer writes  5      15                               — число 4 потеряно
Consumer reads   5                       12
Producer writes  6      21
Producer writes  7      28                               — число 6 потеряно
Consumer reads   7                       19
```

Продолжение ⇨

```
Consumer reads     7                    26 — число 7 прочитано повторно
Consumer reads     7                    33 — число 7 прочитано повторно
Producer writes    8       36
Producer writes    9       45                — число 8 потеряно
Consumer reads     9                    42
Producer writes   10       55
Producer done producing
Terminating Producer
Consumer reads    10                    52
Consumer reads    10                    62 — число 10 прочитано повторно
Consumer read values totaling 62
Terminating Consumer
```

```
Action            Value   Sum of Produced  Sum of Consumed
------            -----   ---------------  ---------------
Consumer reads     -1                      -1 — прочитано ошибочное число –1
                                                (начинать должен производитель)
Producer writes     1       1
Producer writes     2       3                — число 1 потеряно
Consumer reads      2                        1
Consumer reads      2                        3 — число 2 прочитано повторно
Producer writes     3       6
Consumer reads      3                        6
Producer writes     4      10
Consumer reads      4                       10
Consumer reads      4                       14 — число 4 прочитано повторно
Producer writes     5      15
Consumer reads      5                       19
Producer writes     6      21
Consumer reads      6                       25
Producer writes     7      28
Consumer reads      7                       32
Producer writes     8      36
Producer writes     9      45               — число 8 потеряно
Producer writes    10      55               — число 9 потеряно
Producer done producing
Terminating Producer
Consumer reads     10                       42
Consumer read values totaling 42
Terminating Consumer
```

```
Action            Value   Sum of Produced   Sum of Consumed
------            -----   ---------------   ---------------
Producer writes     1       1
Consumer reads      1                         1
Producer writes     2       3
Producer writes     3       6                 — число 2 потеряно
Consumer reads      3                         4
Producer writes     4      10
Consumer reads      4                         8
Consumer reads      4                        12 — число 4 прочитано повторно
Consumer reads      4                        16 — число 4 прочитано повторно
```

```
Consumer reads     4                    20 — число 4 прочитано повторно
Producer writes    5       15
Producer writes    6       21              — число 5 потеряно
Consumer reads     6                    26
Producer writes    7       28
Producer writes    8       36              — число 7 потеряно
Producer writes    9       45              — число 8 потеряно
Consumer reads     9                    35
Producer writes   10       55
Producer done producing
Terminating Producer
Consumer reads    10                    45
Consumer reads    10                    55 — число 10 прочитано повторно
Consumer read values totaling 55           — случайно получилась правильная сумма
Terminating Consumer
```

Илл. 17.6 | Конкурентные потоки jthread используют несинхронизированный буфер (UnsynchronizedBuffer не является потокобезопасным)

Поток-производитель

Поток-производитель выполняет лямбду `produce` (строки 15–36). В каждой итерации цикла поток засыпает на случайное время в диапазоне от `0` до `3000` миллисекунд (строка 27). Когда поток пробуждается, в строке 29 произведенное им число записывается в общий `buffer` путем передачи `count` в функцию `put` этого объекта. В строках 30–31 суммируются все числа, произведенные на данный момент, и эта сумма выводится на экран. После завершения цикла в строке 34 выводится сообщение о том, что производитель выполнил свою задачу и завершает работу. Затем лямбда завершается, и вслед за ней завершается поток-производитель. Любая функция, вызываемая из задачи потока `jthread` (например, функция `put` объекта `buffer`), выполняется в этом потоке. Это будет важно в разделах 17.6–17.7, когда мы реализуем синхронизацию отношений между производителем и потребителем.

Поток-потребитель

Поток-потребитель выполняет лямбду `consume` (строки 39–60). В строках 48–55 выполняется 10 итераций цикла. В каждой итерации поток засыпает на случайное время в диапазоне от 0 до 3000 миллисекунд (строка 51). В строке 53 число читается из `buffer` функцией `get` этого объекта, затем прибавляется к `sum`. В строке 54 выводится на экран сумма всех чисел, потребленных на данный момент. После завершения цикла в строках 57–58 выводится на экран сумма всех потребленных чисел, лямбда завершается и вслед за ней завершается поток-потребитель. После завершения обоих потоков программа завершается.

Генераторы случайных чисел

Генерация случайных чисел в C++11 не является потокобезопасной. Чтобы гарантировать, что каждый поток будет безопасно генерировать случайные числа, мы определили два отдельных генератора случайных чисел в лямбдах `produce`

(строки 18–20) и `consume` (строки 42–44), вместо того чтобы использовать общий для них генератор случайных чисел[1].

Мы вызываем std::this_thread::sleep_for только в демонстрационных целях

На протяжении этой главы мы будем обращаться к **асинхронным (asynchronous)** конкурентным потокам. Когда мы называем их асинхронными, мы имеем в виду, что потоки работают практически независимо друг от друга[2]. Чтобы подчеркнуть, что относительная скорость выполнения асинхронных параллельных потоков непредсказуема, мы вызываем функцию `std::this_thread::sleep_for` в лямбдах `produce` и `consume`. Таким образом, мы не знаем, в какие моменты поток-производитель записывает числа в буфер и в какие моменты поток-потребитель читает числа из буфера. В многопоточных приложениях вообще невозможно предсказать, когда каждый из потоков начнет выполнять свою задачу и сколько времени займет ее выполнение, когда этому потоку будет предоставляться процессор. Эти вопросы планирования потоков контролируются операционной системой.

Если не будет вызовов функции `sleep_for` и если поток-производитель начнет выполняться первым, то с учетом быстродействия современных процессоров поток-производитель, скорее всего, выполнит свою задачу и завершится еще до того, как поток-потребитель получит шанс запуститься. А если первым начнет выполняться поток-потребитель, он, скорее всего, десять раз прочитает одни и те же мусорные данные и завершится еще до запуска потока-производителя.

Анализ выходных данных[3]

Напомним, что поток-производитель должен начинать выполнение первым, а каждое произведенное им значение должно быть использовано потоком-потребителем ровно один раз. Мы выделили в выходных данных те строки, где производитель и потребитель действовали несогласованно, чтобы подчеркнуть проблемы, возникающие без синхронизации доступа к общим изменяемым данным:

- В первом примере выходных данных (илл. 17.6) обратите внимание, что производитель записал в буфер числа `1` и `2` еще до того, как потребитель прочитал свое первое число (`2`). Поэтому число `1` было потеряно. Затем были потеряны числа `4`, `6` и `8`, в то время как `2` и `10` были прочитаны дважды, а `7` — трижды. В этот раз потребитель получил ошибочную сумму `62` вместо `55`.
- Во втором примере потребитель прочитал из буфера мусорное значение `-1` еще до того, как производитель записал в буфер свое первое число. Затем

---

[1] Стандарт C++, «16.5.5.10. Data Race Avoidance». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/res.on.data.races.

[2] Харви Дейтел (Harvey Deitel), Пол Дейтел (Paul Deitel) и Дэвид Чоффнес (David Choffnes), глава 3, «Process Concepts». «Operating Systems», 3-е изд., с. 124. Upper Saddle River, NJ: Prentice Hall, 2004.

[3] Как и в программе на илл. 17.4, фрагменты выходных данных могут чередоваться, из-за чего текст будет выглядеть поврежденным. Для предотвращения этого требуется синхронизация потоков, о которой мы поговорим в разделе 17.6.

были потеряны числа `1`, `8` и `9`, а числа `2` и `4` были прочитаны дважды. В этот раз потребитель получил ошибочную сумму `42`.

- В третьем примере вы видите, что иногда потребитель может прочитать числа, случайно составляющие сумму `55`, даже если каждое число от `1` до `10` не будет прочитано ровно один раз. В этот раз были потеряны числа `2`, `5`, `7` и `8`, число `4` было прочитано четырежды, а число `10` — дважды.

Полученные результаты ясно показывают, что доступ конкурентных потоков к общему для них изменяемому объекту должен тщательно контролироваться. В противном случае программа, скорее всего, выдаст неправильные результаты или, что еще хуже, может случайно выдать «правильный результат» по неправильным причинам.

Одна из больших проблем многопоточного программирования — обнаружение ошибок. Они могут возникать так редко и непредсказуемо, что «сломанная» программа не будет выдавать никаких ошибочных результатов при тестировании, создавая иллюзию правильной работы. По этой причине мы рекомендуем вам пользоваться стандартными контейнерами и примитивами более высокого уровня, которые выполняют синхронизацию без вашего участия.

В следующем разделе мы покажем, как решается проблема потерянных и дублированных данных. В этом примере мы синхронизируем доступ к общему объекту, чтобы гарантировать, что каждое значение будет обработано один и только один раз.

17.6. Синхронизация доступа производителя и потребителя к общим изменяемым данным

Ошибки, продемонстрированные в программе из раздела 17.5, объясняются тем, что класс `UnsynchronizedBuffer` не является потокобезопасным. Он позволяет конкурентным потокам — производителю и потребителю — несогласованно изменять и читать общие данные, что приводит к **«гонкам данных» («data races»)**, или «гоночным состояниям» в программе[1,2]. Тот поток, который выигрывает гонку, вырываясь вперед, выполняет свою задачу, даже если ему нельзя это делать. Не существует гарантии какого-либо определенного порядка выполнения задач конкурентными потоками-производителями и потоками-потребителями. Из-за этого возникают ситуации, в которых:

- производитель записывает новые значения взамен предыдущих, еще не прочитанных потребителем, что приводит к потере данных;

---

[1] Стандарт C++, «Data Races». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/intro.races#def:data_race.

[2] Артур О'Двайр (Arthur O'Dwyer), «Back to Basics: Concurrency», 6 октября 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=F6Ipn7gCOsY.

- потребитель читает ошибочные значения (например, `-1` из строки 23 на илл. 17.5) раньше, чем производитель успеет записать свое первое значение;
- потребитель повторно читает устаревшие значения, уже прочитанные им ранее.

В стандарте C++ подчеркивается, что «невозможен безопасный доступ двух потоков к ячейке памяти без какой-либо формы блокировки, за исключением того случая, когда оба потока имеют доступ только для чтения»[1].

Синхронизация потоков, взаимное исключение и критические секции

Проблемы из последнего примера решаются так: в любой момент времени только один из потоков получает исключительный доступ к той части кода, которая обращается к общим изменяемым данным. В это время другой поток должен ждать. Когда поток с исключительным доступом завершит работу с общими изменяемыми данными, ожидающий поток сможет продолжить свою работу. Такой процесс **синхронизации потоков (thread synchronization)** координирует доступ конкурентных потоков к общим изменяемым данным. Каждый поток, обращаясь к общему объекту, тем самым исключает возможность одновременного обращения другого потока к этому же объекту. Это называется **взаимным исключением (mutual exclusion)**. Те части кода, которые мы защищаем методом взаимного исключения, называются **критическими секциями (critical sections)**.

Атомарность операций чтения и записи

Чтобы сделать наш общий буфер потокобезопасным, мы должны гарантировать, что в любой момент времени только один поток сможет выполнять с этим буфером операцию чтения или записи. Мы также должны гарантировать, что каждая операция чтения или записи будет выполняться полностью, не обрываясь на каком-то из своих промежуточных этапов, — такие операции называются **атомарными (atomic)**. Для этого мы разрешаем только одному потоку в любой момент времени вызывать или функцию `put`, или функцию `get`:

- Если производитель выполняет `put` в тот момент, когда потребитель пытается выполнить `get`, потребитель должен подождать, пока производитель не завершит вызов `put`.
- Аналогично, если потребитель выполняет `get` в тот момент, когда производитель пытается выполнить `put`, производитель должен подождать, пока потребитель не завершит вызов `get`.

Неизменяемые данные не требуют синхронизации

Синхронизация, которую мы демонстрируем в следующем примере, необходима только для общих изменяемых данных, то есть для таких данных, которые могут меняться во время работы программы. Константные данные не изменяются,

[1] «C++11 Language Extensions — Concurrency». Был доступен: 15 сентября 2023 г. https://isocpp.org/wiki/faq/cpp11-language-concurrency.

поэтому доступ к ним можно предоставить любому количеству конкурентных потоков. В C++ Core Guidelines говорится: «При обращении к константам у вас не возникнут гоночные состояния». Поэтому вы должны:

- использовать `const` в определениях объектов со значениями, которые не должны изменяться после создания;
- использовать `constexpr` для значений, которые могут быть вычислены во время компиляции[1].

При соблюдении этих правил значения переменных гарантированно не изменятся после их инициализации, что предотвращает случайные изменения, нарушающие потокобезопасность. В C++ Core Guidelines также рекомендуется как можно меньше использовать общие изменяемые данные, чтобы избежать гоночных состояний[2].

17.6.1. Класс SynchronizedBuffer: мьютексы, блокировки и условные переменные

На илл. 17.7–17.8 показано, как поток-производитель и поток-потребитель корректно обращаются к синхронизированному общему буферу. В этом примере:

- производитель всегда записывает значение первым;
- потребитель корректно читает значение только после того, как производитель его запишет;
- производитель корректно записывает следующее значение только после того, как потребитель прочитает предыдущее (или первое) значение.

Как вы скоро увидите, функции `put` и `get` (илл. 17.7) класса `SynchronizedBuffer` обеспечивают синхронизацию. В демонстрационных целях мы выводим на экран сообщения из этих функций. Операции ввода-вывода выполняются медленно по сравнению с операциями процессора. Из-за медлительности операций ввода-вывода их не следует выполнять в критических секциях кода, так как очень важно минимизировать время, в течение которого объект заблокирован.

```
1 // Fig. 17.7: SynchronizedBuffer.h
2 // Класс SynchronizedBuffer предоставляет производителю и потребителю
3 // синхронизированный доступ к общей изменяемой переменной
4 #pragma once
5 #include <condition_variable>
6 #include <fmt/format.h>
7 #include <mutex>
8 #include <iostream>
9 #include <string>
```

Продолжение ⇨

[1] C++ Core Guidelines, «Con: Constants and Immutability». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-const.

[2] C++ Core Guidelines, «CP.3: Minimize Explicit Sharing of Writable Data». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconc-data.

```
10
11 using namespace std::string_literals;
12
13 class SynchronizedBuffer {
14 public:
15    // Записываем значение в m_buffer
16    void put(int value) {
17       // Критическая секция, требующая блокировки для изменения общих данных
18       {
19          // Блокируем m_mutex, чтобы получить право записи в m_buffer
20          std::unique_lock dataLock{m_mutex};
21
22          if (m_occupied) {
23             std::cout << fmt::format(
24                "Producer tries to write.\n{:<40}{}\t\t{}\n\n",
25                "Buffer full. Producer waits.", m_buffer, m_occupied);
26
27             // Ожидание в зависимости от условной переменной m_cv; лямбда
28             // во втором аргументе гарантирует, что поток продолжит ждать, если
29             // получит процессор раньше, чем m_occupied станет равно false
30             m_cv.wait(dataLock, [&]() {return !m_occupied;});
31          }
32
33          // Запись в m_buffer
34          m_buffer = value;
35          m_occupied = true;
36
37          std::cout << fmt::format("{:<40}{}\t\t{}\n\n",
38             "Producer writes "s + std::to_string(value),
39             m_buffer, m_occupied);
40       } // Деструктор объекта dataLock освобождает блокировку m_mutex
41
42       // Если потребитель ожидает, сообщаем ему, что теперь он может читать
43       m_cv.notify_one();
44    }
45
46    // Возвращаем значение из m_buffer
47    int get() {
48       int value; // Переменная для значения, возвращенного функцией get
49
50       // Критическая секция, требующая блокировки для изменения общих данных
51       {
52          // Блокируем m_mutex, чтобы получить право чтения из m_buffer
53          std::unique_lock dataLock{m_mutex};
54
55          if (!m_occupied) {
56             std::cout << fmt::format(
57                "Consumer tries to read.\n{:<40}{}\t\t{}\n\n",
58                "Buffer empty. Consumer waits.", m_buffer, m_occupied);
59
60             // Ожидание в зависимости от условной переменной m_cv; лямбда
61             // во 2-м аргументе гарантирует, что поток продолжит ждать, если
```

```
62            // получит процессор раньше, чем m_occupied станет равно false
63            m_cv.wait(dataLock, [&]() {return m_occupied;});
64         }
65
66         value = m_buffer;
67         m_occupied = false;
68
69         std::cout << fmt::format("{:<40}{}\t\t{}\n{}\n",
70            "Consumer reads "s + std::to_string(m_buffer),
71            m_buffer, m_occupied, std::string(64, '-'));
72      } // Деструктор объекта dataLock освобождает блокировку m_mutex
73
74      // Если производитель ожидает, сообщаем ему, что он может записывать
75      m_cv.notify_one();
76
77      return value;
78   }
79 private:
80   int m_buffer{-1}; // Общий буфер для производителя и потребителя
81   bool m_occupied{false};
82   std::condition_variable m_cv;
83   std::mutex m_mutex;
84 };
```

Илл. 17.7 | Класс SynchronizedBuffer предоставляет производителю и потребителю синхронизированный доступ к общей изменяемой переменной

Элементы данных m_buffer и m_occupied в SynchronizedBuffer

В строке 80 определен элемент данных `m_buffer` типа `int`, в который поток-производитель будет записывать данные и из которого поток-потребитель будет читать данные. Элемент данных `m_occupied` типа `bool` (строка 81) — это индикатор, указывающий, когда в `m_buffer` находятся актуальные данные. По текущему значению `m_occupied` мы будем отслеживать состояние общего буфера для синхронизации потоков. Переменные `m_occupied` и `m_buffer` относятся к информации о состоянии объекта `SynchronizedBuffer`, поэтому мы должны синхронизировать доступ к ним, чтобы сделать наш буфер потокобезопасным.

Элемент данных типа std::condition_variable в SynchronizedBuffer

Для синхронизации доступа к буферу мы должны гарантировать, что поток-производитель и поток-потребитель выполняют свою работу только тогда, когда буфер находится в нужном состоянии. Нам необходим способ поддерживать потоки в состоянии ожидания в зависимости от двух условий, которые мы отслеживаем с помощью `m_occupied`:

- Производитель может записать новый элемент в буфер только при условии, что буфер не заполнен, то есть `m_occupied` равно `false`.
- Потребитель может прочитать элемент из буфера только при условии, что буфер не пуст, то есть `m_occupied` равно `true`.

Если условие выполняется, соответствующий поток может продолжать работу. Если условие не выполняется, соответствующий поток должен подождать, пока оно не будет выполнено.

Нам также нужен способ уведомлять ожидающий поток об изменении состояния буфера, которое позволяет этому потоку продолжить работу:

- Если потребитель ожидает разрешения на чтение, а производитель уже записал значение в буфер, то буфер заполнен, поэтому производитель должен сообщить ожидающему потребителю, что можно прочитать это значение.
- Если производитель ожидает разрешения на запись, а потребитель уже прочитал значение из буфера, то буфер пуст, поэтому потребитель должен сообщить ожидающему производителю, что можно записать следующее значение.

Эти ожидания и уведомления реализуются с помощью **условной переменной (condition variable)**, которая является объектом класса `std::condition_variable`[1] (заголовок `<condition_variable>`). В строке 82 определен элемент данных `m_cv` этого типа.

Элемент данных типа std::mutex в SynchronizedBuffer

Распространенным способом реализации исключительного доступа к общим изменяемым ресурсам является создание критических секций. Эти синхронизированные блоки кода выполняются атомарно с помощью заголовка `<mutex>`. **Мьютекс (mutex)** — это объект класса `std::mutex`, которым в любой момент времени может владеть только один поток. Поток, которому нужен исключительный доступ к ресурсу, должен прежде всего захватить блокировку мьютекса — обычно это происходит в начале блока кода. Захват блокировки означает блокирование всех остальных потоков, выполняющих операции, для которых нужен этот же мьютекс. Эти потоки остаются заблокированными до тех пор, пока поток-владелец не снимет блокировку — обычно это происходит в конце блока кода. После этого какой-то из ранее заблокированных потоков сможет захватить блокировку и продолжить свою работу[2].

Поместив операции в критическую секцию, мы гарантируем, что в любой момент времени только один поток сможет захватить блокировку и выполнять свои операции. Если несколько критических секций синхронизированы одним и тем же мьютексом, только одна из них может выполняться в любой момент времени. В строке 83 в качестве мьютекса определен элемент данных `m_mutex`, который мы используем

---

[1] «std::condition_variable». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/thread/condition_variable.

[2] Из переписки с Энтони Уильямсом (Anthony Williams) от 6 декабря 2021 г.: «Нет никаких требований к тому, какой поток получает блокировку мьютекса, когда мьютекс разблокирован. <...> с точки зрения ОС во многих случаях наиболее эффективно позволить потоку, который ждал наименьшее время, ... получить блокировку мьютекса, поскольку его данные все еще могут находиться в кэше».

вместе с элементом данных `m_cv` для защиты критических секций кода в функциях `put` и `get` класса `SynchronizedBuffer`, обрабатывающих общие изменяемые данные этого класса. В C++ Core Guidelines рекомендуется определять все данные, к которым нужен исключительный доступ, вместе с мьютексом, защищающим эти данные. Например, мы защитим элементы данных класса `SynchronizedBuffer` мьютексом, который тоже является элементом данных класса `SynchronizedBuffer`[1].

В многопоточных программах размещайте все обращения к общим изменяемым данным внутри критических секций, синхронизированных объектом `std::mutex`, общим для этих данных. Каждая операция внутри критической секции является атомарной операцией. Без промедления снимайте блокировку, когда она больше не нужна.

Функции класса SynchronizedBuffer

Функции `put` (строки 16–44) и `get` (строки 47–78) обеспечивают синхронизированный доступ к общим элементам данных `m_occupied` и `m_buffer`. В любой момент времени только один поток может войти в критические секции этих функций в конкретном объекте `SynchronizedBuffer`.

Переменная `m_occupied` в функциях `put` и `get` всегда показывает, «чей ход»: производителя или потребителя:

- Если `m_occupied` имеет значение `false`, то `m_buffer` пуст, поэтому производитель может записывать в `m_buffer`, но потребитель не может читать из `m_buffer`.
- Если `m_occupied` имеет значение `true`, то `m_buffer` заполнен, поэтому потребитель может читать из `m_buffer`, но производитель не может записывать в `m_buffer`.

Функция put и поток-производитель

Мы синхронизируем доступ к общим изменяемым данным объекта `SynchronizedBuffer` с помощью условной переменной `condition_variable`, мьютекса и **объекта блокировки (lock object)** `std::unique_lock` (C++11). Объекты блокировки нужны для блокировки мьютексов, которые используются совместно с условными переменными `condition_variable`. Когда поток-производитель вызывает функцию `put`, он должен прежде всего захватить блокировку `m_mutex`, чтобы гарантировать свой исключительный доступ к общим изменяемым данным объекта `SynchronizedBuffer`. Для захвата блокировки мьютекса надо создать объект блокировки и инициализировать его мьютексом (строка 20). Если захват блокировки мьютекса в данный момент невозможен, то поток, создающий объект блокировки, блокируется и ждет, пока не получит возможность захватить блокировку. Как только поток захватит блокировку

[1] C++ Core Guidelines, «CP.50: Define a mutex Together with the Data It Guards». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconc-mutex.

мьютекса, вся остальная часть соответствующего блока кода станет **защищенной (guarded)** этим мьютексом.

Скоро вы увидите, что в том случае, когда очередь соответствующего потока еще не наступила, объект блокировки `unique_lock` может снять блокировку, а впоследствии захватить снова. Эта возможность очень важна. Если бы объект `SynchronizedBuffer` удерживал блокировку мьютекса в то время, когда поток не может выполнять свою задачу, программа могла бы оказаться в тупике.

Захват блокировки и проверка разрешения на запись

В строке 20 захватывается блокировка мьютекса `m_mutex`, если эта операция в данный момент доступна. Затем в строке 22 выполняется проверка значения переменной `m_occupied`. Если оно равно `true`, то буфер `m_buffer` заполнен, поэтому поток-производитель должен ждать, пока `m_buffer` не опустеет; в строках 23–25 выводится сообщение о том, что:

- поток-производитель должен записать значение,
- но буфер `m_buffer` заполнен,
- поэтому поток-производитель ждет.

Ожидание в зависимости от значения условной переменной

Условная переменная (объект класса `condition_variable`) нужна, чтобы заставить поток ждать, если не выполнено какое-либо условие, а после выполнения этого условия уведомить ожидающий поток о необходимости продолжить работу:

- Если поток-производитель захватил блокировку мьютекса, но буфер заполнен, этот поток вызывает функцию `wait` класса `condition_variable` (строка 30), передавая ей в качестве первого аргумента объект `unique_lock` с именем `dataLock`. Это приводит к тому, что `dataLock` снимает блокировку мьютекса, поток-производитель переходит в состояние ожидания (*waiting*) в объекте `m_cv` класса `condition_variable` и выводится из соперничества за процессор. Это важно, потому что поток-производитель теперь не может выполнять свою задачу. Чтобы значение `m_occupied` изменилось, поток-потребитель должен получить доступ к `SynchronizedBuffer`. И теперь у потока-потребителя есть возможность захвата блокировки `m_mutex`.
- Когда поток-потребитель выполнит критическую секцию в функции `get`, то есть прочитает значение из буфера, буфер будет считаться пустым. Теперь условие, которого ждет поток-производитель, выполнено. Чтобы сообщить об этом, поток-потребитель вызывает функцию `notify_one` объекта `m_cv` (строка 75). Это позволяет потоку-производителю переключиться из состояния ожидания (*waiting*) в состояние готовности (*ready*)[1]. Когда операционная си-

[1] Из переписки с Энтони Уильямсом (Anthony Williams) от 6 декабря 2021 г.: «Условные переменные не указывают, какой поток будет разбужен при вызове функции notify_one. Это применимо ко *всем* потокам, ожидающим пробуждения. ...В частности, с точки зрения ОС во

стема переключает поток-производитель в состояние выполнения (*running*), он может попытаться снова захватить блокировку[1].

Когда поток-производитель получает сообщение и снова неявно захватывает блокировку `m_mutex`, функция `put` продолжает выполнение со следующей инструкции после вызова функции `wait`.

Ошибочное пробуждение

Иногда операционная система может переключить ожидающий поток в состояние готовности раньше, чем он сможет выполнить свою задачу. Это называется **ложным пробуждением (spurious wakeup)**[2,3]. Второй аргумент функции `wait` в строке 30 — это лямбда без аргумента, которая проверяет, наступила ли очередь потока-производителя обращаться к `m_buffer`. Получив процессор, поток-производитель прежде всего повторно захватывает блокировку `m_mutex`, а затем вызывает эту лямбду. Если лямбда возвращает `false`, поток снимает блокировку и переключается обратно в состояние ожидания. Если лямбда возвращает `true`, поток-производитель продолжает выполнение.

Обновление состояния SynchronizedBuffer

В строке 34 полученное значение присваивается переменной `m_buffer`. Для индикации того, что `m_buffer` теперь содержит значение (которое может прочитать потребитель, но пока не может перезаписать производитель), в строке 35 переменная `m_occupied` переключается в `true`. В строках 37–39 выводится сообщение о том, что производитель записывает значение в `m_buffer`.

Снятие блокировки

Программа подошла к строке 40 — это конец блока, определяющего критическую секцию функции `put`. Когда поток завершает использование общего объекта, управляемого с помощью `unique_lock`, он должен явно или неявно вызвать функцию `unlock` класса `unique_lock`, которая снимает блокировку. Объекты блокировки используют концепцию RAII (раздел 11.5). Если вы не вызовете функцию `unlock` явно, деструктор класса `unique_lock` вызовет ее неявно, когда объект блокировки выйдет из области видимости в конце критической секции. Если поток-потребитель ранее был заблокирован при попытке войти в критическую секцию функ-

---

многих случаях наиболее эффективно позволить разбудить поток, который ждал наименьшее время..., поскольку его данные могут все еще находиться в кэше».

[1] В некоторых приложениях, когда поток снова получает блокировку, он все еще не может выполнить свою задачу. В этом случае он снова переходит в состояние ожидания в условной переменной и неявно освобождает блокировку.

[2] Мариус Банчила (Marius Bancila), «Modern C++ Programming Cookbook», 2-е изд., с. 422. Birmingham, UK: Packt Publishing, 2020 г.

[3] «Spurious Wakeup». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Spurious_wakeup.

ции get, защищенную тем же мьютексом, то теперь поток-потребитель может захватить блокировку для продолжения работы.

Уведомление потока-потребителя о разрешении продолжать работу

Теперь, когда буфер заполнен, поток-производитель вызывает функцию notify_one объекта m_cv (строка 43), чтобы сообщить, что состояние буфера изменилось. Если поток-потребитель был в состоянии ожидания, он переключается в состояние готовности и получает разрешение на повторный захват блокировки. Функция notify_one немедленно возвращает управление, а затем функция put возвращает управление в свою точку вызова (то есть в поток-производитель). Для ускорения работы программы необходимо разблокировать unique_lock перед вызовом notify_one (или notify_all), чтобы ожидающий поток после получения уведомления не продолжал ждать возможности захвата блокировки мьютекса[1].

Функция get и поток-потребитель

Функции get и put реализованы аналогично. Когда поток-потребитель вызывает get, инструкция в строке 53 создает unique_lock и пытается получить блокировку m_mutex. Если блокировка недоступна (например, производитель еще не освободил ее), поток-потребитель блокируется до тех пор, пока блокировка не станет доступной. Если она доступна, она захватывается в строке 53, то есть поток-потребитель теперь владеет блокировкой. Далее в строке 55 проверяется условие: имеет ли m_occupied значение false. Если да, то m_buffer пуст, поэтому в строках 56–58 выводится сообщение о том, что:

- поток-потребитель должен прочитать значение,
- но буфер m_buffer пуст;
- поэтому поток-потребитель ждет.

Ожидание в зависимости от значения условной переменной

В строке 63 вызывается функция wait объекта m_cv, чтобы переключить поток-потребитель в состояние ожидания (*waiting*) в этом объекте. Опять же, wait заставляет dataLock освободить блокировку m_mutex, чтобы поток-производитель мог попытаться получить ее и выполнить свою работу.

Поток-потребитель остается в состоянии ожидания до тех пор, пока не получит уведомление от потока-производителя о необходимости продолжить работу. В этот момент поток-потребитель возвращается в состояние готовности (*ready*). Когда операционная система переводит поток в состояние выполнения (*running*), поток-потребитель снова пытается неявно получить блокировку m_mutex. Если она доступна, поток-потребитель вновь захватывает ее и продолжает выполнение со следующей после wait инструкции. Вторым аргументом функции wait в объекте m_cv является лямбда, которая проверяет, наступила ли очередь потока-по-

[1] Мариус Банчила (Marius Bancila), «Modern C++ Programming Cookbook», с. 420.

требителя. Если поток-потребитель получает процессор и эта лямбда возвращает `false`, поток возвращается в состояние ожидания в `m_cv`.

Обновление состояния SynchronizedBuffer и уведомление производителя о необходимости продолжения работы

В строке 66 сохраняется значение, которое будет возвращено вызывающему потоку. В строке 67 элемент `m_occupied` получает значение `false`, это показывает, что `m_buffer` теперь пуст (то есть производитель может в него записывать), а в строках 69–71 выводится сообщение о том, что потребитель читает значение. В этот момент `dataLock` выходит из области видимости и его деструктор освобождает блокировку `m_mutex`. Далее в строке 75 вызывается функция `notify_one` объекта `m_cv`. Если поток-производитель находится в состоянии ожидания в `m_cv`, он переходит в состояние готовности и получает право на повторный захват блокировки `m_mutex`. Функция `notify_one` немедленно возвращает управление, а затем функция `get` возвращает управление в свою точку вызова.

Согласование ожиданий и уведомлений

Когда конкурентные потоки обращаются к общему объекту, используя блокировки его мьютекса, убедитесь, что в случае вызова одним из потоков функции `wait` для перехода в состояние ожидания в объекте `condition_variable` другой поток в конечном итоге вызовет функцию `notify_one` класса `condition_variable` для перехода ожидающего потока обратно в состояние готовности.

17.6.2. Тестирование класса SynchronizedBuffer

Следующий пример (илл. 17.8) похож на предыдущий (илл. 17.6). В строках 19–24 определяется лямбда `getSleepTime`, которая использует `std::mutex` (строка 16) и `std::lock_guard` (C++11, строка 21) для синхронизированного доступа к генератору случайных чисел, который мы будем делить между потоками. В данном примере мы делаем это только в демонстрационных целях, чтобы показать использование `lock_guard` для защиты общего ресурса, который не требует `condition_variable`. Когда вы создаете `lock_guard`, он захватывает блокировку своего аргумента `mutex`, если она доступна; в противном случае поток, создающий `lock_guard`, блокируется до тех пор, пока не сможет захватить блокировку. В отличие от объекта `unique_lock`, который может освобождать и снова захватывать блокировку мьютекса (для этого необходима условная переменная), `lock_guard` владеет блокировкой мьютекса до тех пор, пока не выйдет из области видимости. В этот момент деструктор `lock_guard` освобождает блокировку мьютекса[1]. Таким образом, в любой момент времени только один поток может выполнять блок `getSleepTime`. Для потоков, которым нужен доступ к ресурсам, защищаемым отдельными мьютексами, в C++ также предусмотрен `std::scoped_lock` (C++17),

[1] «std::lock_guard». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/thread/lock_guard.

который захватывает блокировки сразу нескольких мьютексов и освобождает их все, когда `scoped_lock` выходит из области видимости в конце своего закрывающего блока[1].

В строке 27 создается `SynchronizedBuffer`, который будет общим для двух конкурентных потоков: производителя и потребителя. В строках 30–44 и 47–61 определяются лямбды `produce` и `consume`, которые будут вызываться объектами `jthread` (строки 67–68). Каждая лямбда захватывает `buffer` и `getSleepTime` по ссылке. В строках 63–65 на экран выводятся заголовки столбцов вывода. Когда функция `main` завершается, объекты `jthread` выходят из области видимости, автоматически присоединяя потоки, что завершает их выполнение до завершения программы.

```
 1 // Fig. 17.8: SharedBufferTest.cpp
 2 // Конкурентные потоки корректно обращаются к синхронизированному буферу
 3 #include <chrono>
 4 #include <fmt/format.h>
 5 #include <iostream>
 6 #include <mutex>
 7 #include <random>
 8 #include <thread>
 9 #include "SynchronizedBuffer.h"
10
11 int main() {
12    // Инициализируем генератор случайных чисел
13    std::random_device rd;
14    std::default_random_engine engine{rd()};
15    std::uniform_int_distribution ints{0, 3000};
16    std::mutex intsMutex;
17
18    // Лямбда для синхронизированной генерации случайных интервалов сна
19    auto getSleepTime{
20       [&]() {
21          std::lock_guard lock{intsMutex};
22          return std::chrono::milliseconds{ints(engine)};
23       }
24    };
25
26    // Создаем SynchronizedBuffer для хранения чисел int
27    SynchronizedBuffer buffer;
28
29    // Лямбда-производитель записывает числа от 1 до 10 и суммирует их
30    auto produce{
31       [&buffer, &getSleepTime]() {
32          int sum{0};
33
34          for (int count{1}; count <= 10; ++count) {
```

[1] «std::scoped_lock». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/thread/scoped_lock.

```
35             // Получаем случайный интервал сна и переводим поток в сон
36             std::this_thread::sleep_for(getSleepTime());
37
38             buffer.put(count); // Записываем число в буфер
39             sum += count; // Прибавляем это число к сумме
40          }
41
42          std::cout << "Producer done producing\nTerminating Producer\n";
43       }
44    };
45
46    // Лямбда-потребитель читает числа от 1 и 10 и суммирует их
47    auto consume{
48       [&buffer, &getSleepTime]() {
49          int sum{0};
50
51          for (int count{1}; count <= 10; ++count) {
52             // Получаем случайный интервал сна и переводим поток в сон
53             std::this_thread::sleep_for(getSleepTime());
54
55             sum += buffer.get(); // Читаем число из буфера, прибавляем к sum
56          }
57
58          std::cout << fmt::format("\n{} {}\n{}\n",
59             "Consumer read values totaling", sum, "Terminating Consumer");
60       }
61    };
62
63    std::cout << fmt::format("{:<40}{}\t\t{}\n{:<40}{}\t\t{}\n\n",
64       "Operation", "Buffer", "Occupied",
65       "---------", "------", "--------");
66
67    std::jthread producer{produce}; // Запуск потока-производителя
68    std::jthread consumer{consume}; // Запуск потока-потребителя
69 }
```

```
Operation                               Buffer          Occupied
---------                               ------          --------
Consumer tries to read.
Buffer empty. Consumer waits.           -1              false
Producer writes 1                       1               true
Consumer reads 1                        1               false
------------------------------------------------------------
Producer writes 2                       2               true
Producer tries to write.
Buffer full. Producer waits.            2               true
Consumer reads 2                        2               false
------------------------------------------------------------
Producer writes 3                       3               true
Consumer reads 3                        3               false
------------------------------------------------------------
Consumer tries to read.
```

Окончание ⇨

```
Buffer empty. Consumer waits.          3          false
Producer writes 4                      4          true
Consumer reads 4                       4          false
-------------------------------------------------------------
Producer writes 5                      5          true
Consumer reads 5                       5          false
-------------------------------------------------------------
Producer writes 6                      6          true
Producer tries to write.
Buffer full. Producer waits.           6          true
Consumer reads 6                       6          false
-------------------------------------------------------------
Producer writes 7                      7          true
Producer tries to write.
Buffer full. Producer waits.           7          true
Consumer reads 7                       7          false
-------------------------------------------------------------
Producer writes 8                      8          true
Producer tries to write.
Buffer full. Producer waits.           8          true
Consumer reads 8                       8          false
-------------------------------------------------------------
Producer writes 9                      9          true
Consumer reads 9                       9          false
-------------------------------------------------------------
Producer writes 10                     10         true
Producer done producing
Terminating Producer
Consumer reads 10                      10         false
-------------------------------------------------------------
Consumer read values totaling 55
Terminating Consumer
```

Илл. 17.8 | Конкурентные потоки корректно обращаются к синхронизированному буферу

Анализ выходных данных

Изучите выходные данные на илл. 17.8 и обратите внимание:

- Каждое записанное в буфер число читается ровно один раз: никакие числа не теряются и никакие числа не читаются более одного раза.
- Синхронизация гарантирует, что производитель записывает данные только в пустой буфер, а потребитель записывает данные только из заполненного буфера.
- Производитель всегда предоставляет данные до того, как потребитель впервые имеет возможность их получить.
- Потребитель ждет, если производитель не предоставил новые данные с момента последнего получения потребителем.

- Производитель ждет, если потребитель еще не получил данные, предоставленные производителем.

Выполните эту программу несколько раз, чтобы убедиться, что каждое записанное в буфер число будет прочитано из буфера ровно один раз. В выходных данных мы выделили жирным шрифтом строки, показывающие, когда производитель и потребитель ждут своей очереди.

Примечание об инструкциях вывода на экран в наших примерах синхронизации

Наши синхронизированные функции `put` и `get` не только выполняют операции с буфером `SynchronizedBuffer`, но и выводят сообщения о том, что происходит в потоках при выполнении этих функций. Это нужно только для того, чтобы продемонстрировать, что функции `put` и `get` правильно синхронизированы, поскольку сообщения выводятся в правильном порядке. В последующих примерах мы продолжим выводить сообщения из критических секций кода, делая это только в демонстрационных целях.

В C++ Core Guidelines рекомендуется минимизировать продолжительность критических секций кода[1], то есть время, в течение которого объект «заблокирован». Избегайте операций ввода-вывода, длительных вычислений и не требующих синхронизации операций, пока удерживаете блокировку.

Кроме того, в демонстрационных целях мы вызываем функцию `sleep_for` (строки 36 и 53), чтобы подчеркнуть непредсказуемость планирования потоков. Важно отметить, что, в отличие от этого учебного примера, в реальном приложении поток никогда не должен спать, удерживая блокировку.

17.7. Производитель и потребитель: минимизация ожидания с помощью кольцевого буфера

В программе из раздела 17.6 синхронизация потоков гарантировала, что два конкурентных потока правильно обращаются к данным в общем буфере. Однако такие программы могут работать неоптимально. Если потоки выполняются с разной скоростью, один из них будет проводить в ожидании больше времени, чем другой (или даже большую часть времени). Например:

- Если поток-производитель производит значения быстрее, чем потребитель может их потреблять, потоку-производителю приходится ждать, потому что нет места для их записи.

[1] C++ Core Guidelines, «CP.43: Minimize Time Spent in a Critical Section». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconc-time.

- Если потребитель потребляет значения быстрее, чем производитель их производит, потребителю приходится ждать, пока производитель не поместит следующее значение в общий буфер. Даже потоки, работающие с примерно одинаковой относительной скоростью, могут иногда «рассинхронизироваться» на некоторое время, то есть один из них будет долго ждать другой.

Нельзя предсказать относительные скорости выполнения асинхронных параллельных потоков. Взаимодействие с операционной системой, сетью, пользователем и другими компонентами может привести к тому, что потоки будут работать с разными и постоянно меняющимися скоростями. Когда это происходит, одни потоки ждут другие. Если потоки ждут слишком долго, код работает менее эффективно, интерактивные интерфейсы начинают «тормозить», а приложения могут «зависать» из-за потенциально длительных задержек.

Кольцевые буферы

Используя **кольцевой буфер (circular buffer)**, вы можете минимизировать периоды ожидания для конкурентных потоков, которые используют общие ресурсы и работают с одинаковой средней скоростью. Такой буфер состоит из фиксированного количества ячеек (элементов), в которые производитель может записывать значения и из которых потребитель может считывать эти значения. Кольцевой буфер управляет операциями записи производителя в эти ячейки и операциями чтения потребителя из этих ячеек. Все операции выполняются последовательно: с первой ячейки и до последней, затем снова с первой — и так далее по кругу, поэтому буфер называется кольцевым.

Пример взаимодействия производителя и потребителя через кольцевой буфер — это воспроизведение потокового видео, о котором мы говорили в разделе 17.1. Благодаря кольцевому буферу:

- Когда производитель временно работает быстрее потребителя, он может записывать очередные значения в буфер, пока в нем остаются свободные ячейки. Поэтому производитель продолжает работу, даже если потребитель не готов прочитать текущее значение.
- Когда потребитель временно работает быстрее производителя, он может читать очередные значения из буфера, если они там накопились. Поэтому потребитель продолжает работу, даже если производитель не готов записать очередное значение.

Даже с кольцевым буфером производитель может заполнить буфер и будет вынужден ждать, пока потребитель не прочитает значение, чтобы освободить место в буфере. Аналогично, если буфер пуст, потребитель должен ждать, пока производитель не запишет значение.

Кольцевой буфер не помогает, если производитель и потребитель постоянно работают со значительно отличающимися средними скоростями:

- Если потребитель всегда выполняет свою задачу быстрее, достаточно буфера, содержащего одно значение (или небольшое количество значений).
- Если производитель в среднем выполняет свою задачу быстрее, а программа не просит его подождать, то только буфер с «бесконечным» количеством ячеек мог бы хранить избыточные значения.

Но если оба потока выполняют свои задачи примерно с одинаковой средней скоростью, кольцевой буфер помогает сгладить эффект периодического ускорения или замедления любого из потоков и сокращает время ожидания, повышая быстродействие.

Важно оптимизировать размер кольцевого буфера таким образом, чтобы время ожидания потоков было минимальным, но без излишнего расходования памяти. Если мы выясним, что производитель часто создает в три раза больше значений, чем потребитель может прочитать, мы можем сделать буфер из трех ячеек (как минимум), достаточный для таких ситуаций. Если сделать буфер слишком маленьким, это приведет к тому, что потоки будут ждать дольше.

Реализация кольцевого буфера

На илл. 17.9–17.10 показана программа с конкурентными потоками, производителем и потребителем, которые обращаются к кольцевому буферу с синхронизацией (илл. 17.9). Кольцевой буфер реализован в виде массива из трех элементов `int`. Потребитель читает значение, только если массив не пуст, а производитель записывает значение, только если массив не заполнен. Опять же, инструкции вывода на экран, используемые в критических разделах этого класса, предназначены только для демонстрации.

```
 1 // Fig. 17.9: CircularBuffer.h
 2 // Синхронизированный доступ к общему трехэлементному кольцевому буферу
 3 #pragma once
 4 #include <array>
 5 #include <condition_variable>
 6 #include <fmt/format.h>
 7 #include <mutex>
 8 #include <iostream>
 9 #include <string>
10 #include <string_view>
11
12 using namespace std::string_literals;
13
14 class CircularBuffer {
15 public:
16    // Записываем value в m_buffer
17    void put(int value) {
18       // Критическая секция, требующая блокировки для изменения общих данных
19       {
20          // Блокируем m_mutex, чтобы получить право записи в m_buffer
21          std::unique_lock dataLock{m_mutex};
```

Продолжение ⇨

```
22
23          // Если нет места, ждем по значению условной переменной m_cv
24          if (m_occupiedCells == m_buffer.size()) {
25             std::cout << "Buffer is full. Producer waits.\n\n";
26
27             // Ждем по значению условной переменной; лямбда в аргументе
28             // гарантирует, что поток продолжит ждать, если получит
29             // процессор раньше, чем в m_buffer освободится место
30             m_cv.wait(dataLock,
31                [&] {return m_occupiedCells < m_buffer.size();});
32          }
33
34          m_buffer[m_writeIndex] = value; // Записываем в m_buffer
35          ++m_occupiedCells; // Заполняем еще один элемент m_buffer
36          m_writeIndex = (m_writeIndex + 1) % m_buffer.size();
37          displayState(fmt::format("Producer writes {}", value));
38       } // Здесь деструктор объекта dataLock снимает блокировку m_mutex
39
40       m_cv.notify_one(); // Уведомляем потоки, ожидающие права чтения
41    }
42
43    // Читаем value из m_buffer
44    int get() {
45       int readValue; // Для временного хранения прочитанного значения
46
47       // Критическая секция, требующая блокировки для изменения общих данных
48       {
49          // Блокируем m_mutex, чтобы получить право чтения из m_buffer
50          std::unique_lock dataLock{m_mutex};
51
52          // Если нет данных для чтения, переводим поток в состояние ожидания
53          if (m_occupiedCells == 0) {
54             std::cout << "Buffer is empty. Consumer waits.\n\n";
55
56             // Ждем по значению условной переменной; лямбда в аргументе
57             // гарантирует, что поток продолжит ждать, если получит
58             // процессор раньше, чем в m_buffer появятся данные
59             m_cv.wait(dataLock, [&]() {return m_occupiedCells > 0;});
60          }
61
62          readValue = m_buffer[m_readIndex]; // Читаем из m_buffer
63          m_readIndex = (m_readIndex + 1) % m_buffer.size();
64          --m_occupiedCells; // Освобождаем один элемент m_buffer
65          displayState(fmt::format("Consumer reads {}", readValue));
66       } // Здесь деструктор объекта dataLock снимает блокировку m_mutex
67
68       m_cv.notify_one(); // Уведомляем потоки, ожидающие права записи
69       return readValue;
70    }
71
72    // Выводим информацию о текущей операции и о состоянии m_buffer
73    void displayState(std::string_view operation) const {
```

```
74       std::string s;
75
76       // Добавляем аргумент операции и количество занятых элементов m_buffer
77       s += fmt::format("{} (buffer cells occupied: {})\n{:<15}",
78          operation, m_occupiedCells, "buffer cells:");
79
80       // Добавляем данные, которые находятся в m_buffer
81       for (int value : m_buffer) {
82          s += fmt::format(" {:2d} ", value);
83       }
84
85       s += fmt::format("\n{:<15}", ""); // Отступ
86
87       // Добавляем горизонтальные разделители
88       for (int i{0}; i < m_buffer.size(); ++i) {
89          s += "---- ";
90       }
91
92       s += fmt::format("\n{:<15}", ""); // Отступ
93
94       for (int i{0}; i < m_buffer.size(); ++i) {
95          s += fmt::format(" {}{} ",
96             (i == m_writeIndex ? 'W' : ' '),
97             (i == m_readIndex ? 'R' : ' '));
98       }
99
100       s += "\n\n";
101       std::cout << s; // Выводим информацию о состоянии
102    }
103 private:
104    std::condition_variable m_cv;
105    std::mutex m_mutex;
106
107    std::array<int, 3> m_buffer{-1, -1, -1}; // Общий буфер m_buffer
108    int m_occupiedCells{0}; // Счетчик элементов, заполненных данными
109    int m_writeIndex{0}; // Индекс элемента для следующей операции записи
110    int m_readIndex{0}; // Индекс элемента для следующей операции чтения
111 };
```

Илл. 17.9 | Синхронизированный доступ к общему трехэлементному кольцевому буферу

Элементы данных

В строках 104–110 определены элементы данных класса:

- В строках 104–105 определены элементы `m_cv` (`std::condition_variable`) и `m_mutex` (`std::mutex`), необходимые для синхронизации доступа к другим элементам данных `CircularBuffer`.
- В строке 107 создается и инициализируется `m_buffer` — трехэлементный массив чисел `int`, который служит кольцевым буфером.

- Переменная `m_occupiedCells` (строка 108) подсчитывает количество элементов `m_buffer`, содержащих данные для чтения. Когда `m_occupiedCells` равна `0`, кольцевой буфер пуст и потребитель должен ждать. Когда `m_occupiedCells` равна `3` (размер буфера), кольцевой буфер заполнен и производитель должен ждать.
- Переменная `m_writeIndex` (строка 109) указывает позицию, в которую производитель может записать следующее значение.
- Переменная `m_readIndex` (строка 110) указывает позицию, из которой потребитель может прочитать следующее значение.

Элементы данных `m_buffer`, `m_occupiedCells`, `m_writeIndex` и `m_readIndex` относятся к общим изменяемым данным класса, поэтому доступ к этим переменным должен быть синхронизирован для обеспечения потокобезопасности класса `CircularBuffer`.

Функция put класса CircularBuffer

Функция `put` (строки 17–41) выполняет те же задачи, что и на илл. 17.7, но с несколькими изменениями. В строках 24–32 проверяется, заполнен ли `CircularBuffer`. Если да, то производитель должен подождать, поэтому в строке 25 выводится на экран соответствующее сообщение. Затем в строках 30–31 вызывается функция `wait` объекта `m_cv`, заставляющая поток-производитель освободить блокировку `m_mutex` и ждать, пока в буфере не освободится место для записи нового значения.

Когда выполнение продолжается в строке 34, производитель помещает значение в позицию `m_writeIndex` кольцевого буфера. В строке 35 увеличивается значение `occupiedCells`, поскольку буфер уже содержит значение, которое может прочитать потребитель. Затем в строке 36 обновляется `m_writeIndex` для следующего вызова функции `put` производителя. Эта строка реализует цикличность буфера: когда `writeIndex` увеличивается до конца буфера, она устанавливает его обратно в `0`. Далее в строке 37 вызывается функция `displayState` (строки 73–102), чтобы вывести на экран записанное производителем число, состояние счетчика `occupiedCells`, содержимое ячеек буфера и текущие значения `m_writeIndex` и `m_readIndex`. Достижение конца блока в строке 38 освобождает блокировку `m_mutex`. В строке 40 вызывается функция `notify_one` объекта `m_cv` для перехода ожидающего потока в состояние готовности, чтобы ожидающий поток-потребитель (если он есть) мог теперь снова попытаться прочитать значение из буфера.

Функция get класса CircularBuffer

Функция `get` (строки 44–70) выполняет те же задачи, что и на илл. 17.7, но с несколькими незначительными изменениями. В строках 53–60 (илл. 17.9) проверяется, все ли ячейки буфера пусты. Если это так, потребитель должен подождать. При этом в строке 54 выводится сообщение о том, что потребитель ждет из-за отсутствия данных. Затем в строке 59 вызывается функция `wait` объекта `m_cv`, заставляющая поток-потребитель освободить блокировку `m_mutex` и ждать, пока данные не станут доступны для чтения.

Когда выполнение продолжается в строке 62, локальной переменной `readValue` присваивается значение, взятое из позиции `m_readIndex` буфера `m_buffer`. Затем в строке 63 обновляется `m_readIndex` для следующего вызова функции `get`. В этой строке и в строке 36 реализована цикличность буфера. В строке 64 уменьшается `m_occupiedCells`, потому что теперь есть свободная ячейка буфера, в которую производитель может записать число. В строке 65 на экран выводится информация: прочитанное число, количество занятых ячеек буфера, содержимое буфера и текущие значения `m_writeIndex` и `m_readIndex`. Достижение конца блока в строке 66 освобождает блокировку `m_mutex`. В строке 68 вызывается функция `notify_one` объекта `m_cv`, чтобы разрешить ожидающему производителю повторить попытку записи. Затем в строке 69 прочитанное число возвращается в точку вызова.

Функция displayState класса CircularBuffer

Функция `displayState` (строки 73–102) составляет и выводит на экран строку с данными о состоянии программы. В строках 81–83 к этим данным добавляются значения ячеек буфера с помощью спецификатора формата `":2d"`, который форматирует содержимое каждой ячейки с добавлением пробела, если это одна цифра. В строках 94–98 добавляются текущие значения `m_writeIndex` и `m_readIndex` с буквами `W` и `R` соответственно. Обратите внимание, что мы вызываем функцию `displayState` только из критических секций для обеспечения потокобезопасности. Напомним, что в обычных программах нельзя выполнять ввод-вывод в критических секциях, а в данном случае мы делаем это только для получения полезных результатов в демонстрационных целях.

Тестирование класса CircularBuffer

Код, показанный на илл. 17.10, содержит функцию `main`, которая запускает приложение. В строке 13 создается объект `CircularBuffer` с именем `buffer`, а в строке 14 отображается его начальное состояние. В строках 17–37 и 40–60 определены лямбды `produce` и `consume`, которые будут выполняться параллельными потоками `producer` и `consumer`. В строках 62–63 создаются объекты `std::jthread`, которые вызывают лямбды `produce` и `consume`. Когда `main` завершается, объекты `std::jthread` выходят из области видимости, автоматически присоединяя потоки.

```
 1 // Fig. 17.10: SharedBufferTest.cpp
 2 // Конкурентные потоки обращаются к синхронизированному кольцевому буферу
 3 #include <chrono>
 4 #include <fmt/format.h>
 5 #include <iostream>
 6 #include <mutex>
 7 #include <random>
 8 #include <thread>
 9 #include "CircularBuffer.h"
10
11 int main() {
12    // Создаем кольцевой буфер для чисел int и отображаем начальное состояние
13    CircularBuffer buffer;
```

Продолжение ⇨

```
   buffer.displayState("Initial State");

   // Лямбда-выражение производит числа от 1 до 10 и суммирует их
   auto produce{
      [&buffer]() {
         // Инициализируем генератор случайных чисел
         std::random_device rd;
         std::default_random_engine engine{rd()};
         std::uniform_int_distribution ints{0, 3000};

         int sum{0};

         for (int count{1}; count <= 10; ++count) {
            // Получаем случайный интервал сна, затем переводим поток в сон
            std::chrono::milliseconds sleepTime{ints(engine)};
            std::this_thread::sleep_for(sleepTime);

            buffer.put(count); // Записываем число в буфер
            sum += count; // Прибавляем это число к сумме
         }

         std::cout << "Producer done producing\nterminating Producer\n\n";
      }
   };

   // Лямбда-выражение потребляет числа от 1 до 10 и суммирует их
   auto consume{
      [&buffer]() {
         // Инициализируем генератор случайных чисел
         std::random_device rd;
         std::default_random_engine engine{rd()};
         std::uniform_int_distribution ints{0, 3000};

         int sum{0};

         for (int count{1}; count <= 10; ++count) {
            // Получаем случайный интервал сна, затем переводим поток в сон
            std::chrono::milliseconds sleepTime{ints(engine)};
            std::this_thread::sleep_for(sleepTime);

            sum += buffer.get(); // Читаем число из буфера, прибавляем к sum
         }

         std::cout << fmt::format("{} {}\n{}\n\n",
            "Consumer read values totaling", sum, "Terminating Consumer");
      }
   };

   std::jthread producer{produce}; // Запускаем поток-производитель
   std::jthread consumer{consume}; // Запускаем поток-потребитель
}
```

```
Initial State (buffer cells occupied: 0)
buffer cells:   -1   -1   -1
```

```
              ---- ---- ----
               WR

Buffer is empty. Consumer waits.

Producer writes 1 (buffer cells occupied: 1)
buffer cells:    1   -1   -1
              ---- ---- ----
                R   W
Consumer reads 1 (buffer cells occupied: 0)
buffer cells:    1   -1   -1
              ---- ---- ----
                    WR

Buffer is empty. Consumer waits.

Producer writes 2 (buffer cells occupied: 1)
buffer cells:    1    2   -1
              ---- ---- ----
                     R   W
Consumer reads 2 (buffer cells occupied: 0)
buffer cells:    1    2   -1
              ---- ---- ----
                         WR

Buffer is empty. Consumer waits.

Producer writes 3 (buffer cells occupied: 1)
buffer cells:    1    2    3
              ---- ---- ----
               W              R
Consumer reads 3 (buffer cells occupied: 0)
buffer cells:    1    2    3
              ---- ---- ----
               WR
Producer writes 4 (buffer cells occupied: 1)
buffer cells:    4    2    3
              ---- ---- ----
                R   W
Producer writes 5 (buffer cells occupied: 2)
buffer cells:    4    5    3
              ---- ---- ----
                R             W
Consumer reads 4 (buffer cells occupied: 1)
buffer cells:    4    5    3
              ---- ---- ----
                       R   W
Consumer reads 5 (buffer cells occupied: 0)
buffer cells:    4    5    3
              ---- ---- ----
                         WR
Producer writes 6 (buffer cells occupied: 1)
buffer cells:    4    5    6
```

Окончание ⇨

```
                ---- ---- ----
                 W         R
Producer writes 7 (buffer cells occupied: 2)
buffer cells:    7    5    6
                ---- ---- ----
                      W    R
Producer writes 8 (buffer cells occupied: 3)
buffer cells:    7    8    6
                ---- ---- ----
                           WR
Buffer is full. Producer waits.
Consumer reads 6 (buffer cells occupied: 2)
buffer cells:    7    8    6
                ---- ---- ----
                  R        W
Producer writes 9 (buffer cells occupied: 3)
buffer cells:    7    8    9
                ---- ---- ----
                 WR

Buffer is full. Producer waits.

Consumer reads 7 (buffer cells occupied: 2)
buffer cells:    7    8    9
                ---- ---- ----
                 W    R
Producer writes 10 (buffer cells occupied: 3)
buffer cells:   10    8    9
                ---- ---- ----
                      WR

Producer done producing
Terminating Producer

Consumer reads 8 (buffer cells occupied: 2)
buffer cells:   10    8    9
                ---- ---- ----
                      W    R
Consumer reads 9 (buffer cells occupied: 1)
buffer cells:   10    8    9
                ---- ---- ----
                  R   W
Consumer reads 10 (buffer cells occupied: 0)
buffer cells:   10    8    9
                ---- ---- ----
                      WR
Consumer read values totaling 55
Terminating Consumer
```

Илл. 17.10 | Конкурентные потоки обращаются к синхронизированному кольцевому буферу

Анализ выходных данных

Когда производитель записывает число или потребитель читает число, программа выводит сообщение с указанием выполненного действия, содержимого `m_buffer` и позиций `m_writeIndex` (`"W"`) и `m_readIndex` (`"R"`). В приведенном примере выходных данных потребитель сразу переключается в режим ожидания, потому что он не смог обратиться к буферу до того, как производитель выполнил свое первое действие. Затем производитель записывает число 1. После этого буфер содержит 1 в первой ячейке и -1 (значение по умолчанию, которое мы используем для вывода) в двух других ячейках. Индекс записи теперь указывает на вторую ячейку, а индекс чтения — на первую. Далее потребитель читает число 1. Буфер содержит те же значения, но индекс чтения теперь указывает на вторую ячейку. Затем потребитель снова пытается читать, но буфер пуст, поэтому потребитель должен подождать. Потребитель снова ждет после чтения числа 2. Далее выходные данные показывают, что производитель дважды заполняет буфер и каждый раз ожидает.

17.8. Читатели и писатели

В предыдущих примерах мы работали с одним потоком-производителем и одним потоком-потребителем — это характерно для многих приложений с потоками, совместно использующими изменяемые данные. Но в некоторых системах требуется несколько читающих данные потоков-потребителей, которые называются **читателями (readers)**, и несколько записывающих данные потоков-производителей, которые называются **писателями (writers)**. В таких системах возникает «проблема читателей и писателей».

Например, в системе бронирования авиабилетов может быть гораздо больше читателей, чем писателей. Прежде чем клиент приобретет конкретное место на конкретном рейсе, будет сделано множество запросов к базе данных с информацией о доступных рейсах. Важно отметить, что многие читатели могут обращаться к базе данных одновременно без нарушения потокобезопасности, поскольку они не изменяют данные. Но для работы каждого писателя необходим исключительный доступ к критической секции, изменяющей данные, — в это время там не должно быть никаких других писателей и никаких читателей.

В C++ есть классы для создания программ с множеством читателей и писателей: `std::shared_mutex`, `std::shared_lock` и `std::condition_variable_any`. **Разделяемый мьютекс (shared mutex)** класса `std::shared_mutex`[1] (заголовок `<mutex>`) позволяет одному писателю или нескольким читателям владеть его блокировкой:

- Если блокировка доступна, писатель захватывает ее с помощью `lock_guard` или `unique_lock`. В противном случае писатель блокируется и должен ждать, пока блокировка не станет доступной. Пока писатель удерживает блокировку `shared_mutex`, другие потоки не могут ее получить.

[1] «std::shared_mutex». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/thread/shared_mutex.

- Если блокировка доступна, читатель захватывает ее через `std::shared_lock`[1] (заголовок `<mutex>`); в противном случае читатель блокируется и должен ждать, пока блокировка не станет доступной. Пока читатель удерживает блокировку `shared_mutex`, только другие читатели могут ее получить.

Как и в приведенных примерах синхронизированного взаимодействия производителя и потребителя, необходимо убедиться, что читатели и писатели выполняют свои задачи только тогда, когда наступает их очередь:

- Если читатель читает данные, когда прибыл писатель, то писатель должен подождать, пока не станет доступной блокировка. Кроме того, все дополнительные читатели, которые появятся впоследствии, должны будут подождать, пока ожидающий в данный момент писатель не выполнит свою работу и не сообщит им, что он закончил запись. Ожидающий писатель был бы отложен на неопределенное время, если бы потоку прибывающих читателей было разрешено читать.
- Если писатель записывает данные, когда прибыл читатель, то читатель должен подождать, пока не станет доступной блокировка. Кроме того, все дополнительные писатели, которые появятся впоследствии, должны будут подождать, пока ожидающие в данный момент читатели не завершат свою работу. Ожидающие читатели были бы отложены на неопределенное время, если бы потоку прибывающих писателей было разрешено записывать.

И снова мы можем использовать условные переменные для того, чтобы обеспечить выполнение этих условий: одну для читателей, другую для писателей.

Читатели используют объекты `shared_lock`, но классу `condition_variable` необходимы объекты `unique_lock`. Поэтому для управления ожидающими читателями используйте объект `std::condition_variable_any`[2]. Класс `condition_variable_any` работает подобно классу `condition_variable`, но поддерживает другие типы блокировок. Когда поток-писатель заканчивает записывать данные, он вызывает функцию `notify_all` объекта `condition_variable_any`, уведомляя всех ожидающих читателей, что пришло время читать данные. Все потоки-читатели, ожидающие этого условия, перейдут в состояние готовности и получат право на повторное получение блокировки.

Потоки-писатели (один или несколько) по-прежнему будут использовать `unique_lock` для исключительного доступа к общим изменяемым данным, поэтому ожидающими писателями можно управлять с помощью `condition_variable`. Когда все активные читатели закончат чтение, программа вызовет функцию `notify_all` объекта `condition_variable`. Все ожидающие потоки-писатели перейдут в состояние готовности и получат право на повторное получение блокировки. Но только один из них снова получит блокировку и продолжит работу.

---

[1] «std::shared_lock». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/thread/shared_lock.

[2] «std::condition_variable». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/thread/condition_variable_any.

17.9. Кооперативное прерывание потоков класса jthread

Перед завершением работы многопоточного приложения рекомендуется завершить потоки, продолжающие выполнение задач, чтобы освободить используемые этими потоками ресурсы[1]. Например, вам может понадобиться способ закрыть сразу все открытые файлы, соединения с базой данных и сетевые соединения. До C++20 не существовало стандартного механизма взаимодействия потоков для такого корректного завершения. Это было одним из недостатков потоков.

Для решения этой проблемы в C++20 введены средства **кооперативного прерывания (cooperative cancellation)** потоков, позволяющие программе уведомлять потоки о том, что им пора завершиться. Задачи, выполняемые в потоках, могут следить за такими уведомлениями, а при их получении завершать критически важные операции, освобождать ресурсы и прекращать работу.

На илл. 17.11 демонстрируется кооперативное прерывание потоков с помощью средств из заголовка `<stop_token>`. Как вы увидите, эти средства интегрированы с классом `jthread`.

```
 1 // Fig. 17.11: CooperativeCancelation.cpp
 2 // Запрос на прерывание потока std::jthread
 3 // с помощью интегрированного с ним объекта stop_source
 4 #include <chrono>
 5 #include <fmt/format.h>
 6 #include <iostream>
 7 #include <mutex>
 8 #include <random>
 9 #include <sstream>
10 #include <string>
11 #include <string_view>
12 #include <thread>
13
14 // Получаем идентификатор текущего потока в виде строки
15 std::string id() {
16    std::ostringstream out;
17    out << std::this_thread::get_id();
18    return out.str();
19 }
20
21 int main() {
22    // printTask повторяется до получения другим потоком запроса на прерывание
23    auto printTask{
24       [&](std::stop_token token, std::string name) {
25          // Инициализируем генератор случайных чисел
26          std::random_device rd;
```

Продолжение ⇨

[1] Энтони Уильямс (Anthony Williams), «Concurrency in C++20 and Beyond», 16 октября 2019 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=jozHW_B3D4U.

```
        std::default_random_engine engine{rd()};
        std::uniform_int_distribution ints{500, 1000};

        // Регистрируем функцию, вызываемую при запросе на прерывание
        std::stop_callback callback(token, [&]() {
        std::cout << fmt::format(
           "{} told to stop by thread with id {}\n", name, id());
        });

        while (!token.stop_requested()) { // Цикл до запроса на прерывание
           auto sleepTime{std::chrono::milliseconds{ints(engine)}};

           std::cout << fmt::format(
              "{} (id: {}) going to sleep for {} ms\n",
              name, id(), sleepTime.count());

           // Переводим поток в сон на sleepTime миллисекунд
           std::this_thread::sleep_for(sleepTime);

           // Показываем, что задача проснулась
           std::cout << fmt::format("{} working.\n", name);
        }

        std::cout << fmt::format("{} terminating.\n", name);
     }
  };

  std::cout << fmt::format("MAIN (id: {}) STARTING TASKS\n", id());

  // Создаем два jthread, вызывающих printTask со строковым аргументом
  std::jthread task1{printTask, "Task 1"};
  std::jthread task2{printTask, "Task 2"};

  // Переводим главный поток в сон на 2 секунды
  std::cout << "\nMAIN GOING TO SLEEP FOR 2 SECONDS\n\n";
  std::this_thread::sleep_for(std::chrono::seconds{2});

  std::cout << fmt::format("\nMAIN (id: {}) ENDS\n\n", id());
}
```

```
MAIN (id: 16352) STARTING TASKS
MAIN GOING TO SLEEP FOR 2 SECONDS
Task 1 (id: 14048) going to sleep for 708 ms
Task 2 (id: 10504) going to sleep for 995 ms
Task 1 working.
Task 1 (id: 14048) going to sleep for 940 ms
Task 2 working.
Task 2 (id: 10504) going to sleep for 926 ms
Task 1 working.
Task 1 (id: 14048) going to sleep for 875 ms
Task 2 working.
Task 2 (id: 10504) going to sleep for 519 ms
```

```
MAIN (id: 16352) ENDS
Task 2 told to stop by thread with id 16352
Task 2 working.
Task 2 terminating.
Task 1 told to stop by thread with id 16352
Task 1 working.
Task 1 terminating.
```

Илл. 17.11 | Запрос на прерывание потока std::jthread с помощью интегрированного с ним объекта stop_source

Каждый поток `jthread` интегрирован с объектом `std::stop_source`, который, в свою очередь, связан с объектом `std::stop_token`. Функция задачи потока `jthread` опционально может получить `stop_token` в качестве первого параметра. Затем функция задачи может периодически вызывать функцию `stop_requested` объекта `std::stop_token`, проверяя, должна ли задача прервать выполнение. Когда

- другой поток вызывает функцию `request_stop` объекта `jthread`
- или деструктор объекта `jthread` вызывает функцию `request_stop` при выходе `jthread` из области видимости,

объект `stop_source` уведомляет связанный с ним `stop_token` о том, что поступил запрос на прерывание. Последующие вызовы функции `stop_requested` объекта `stop_token` вернут `true`. После этого задача может корректно завершиться. Если функция задачи никогда не вызывает функцию `stop_requested`, соответствующий `jthread` продолжает выполнение — отсюда и название «кооперативное прерывание».

В этом примере мы позволим деструктору объекта `jthread` вызвать функцию `request_stop`, что позволит программе завершиться вскоре после завершения `main`. Мы создаем два потока `jthread` (строки 57–58), которые вызывают лямбду `printTask` (строки 23–52). Каждый `jthread` передает лямбде свой `stop_token`. В этой лямбде есть цикл (строки 36–48), который непрерывно:

- выводит на экран имя задачи, идентификатор потока и интервал сна,
- переводит поток в сон,
- затем снова выводит на экран имя задачи и сообщает, что лямбда выполняет работу.

Цикл выполняется до тех пор, пока объект `stop_source` соответствующего потока `jthread` не получит вызов своей функции `request_stop`. В данном случае этот вызов происходит в деструкторах объектов `jthread`, когда они выходят из области видимости в конце `main`.

Опциональная функция stop_callback

Строки 31–34 также демонстрируют, что вы можете зарегистрировать опциональную функцию, которая будет вызвана при запросе на прерывание потока. Кон-

структор `std::stop_callback`[1] получает в качестве аргументов объект `stop_token` и функцию без параметров, в данном случае лямбду. Конструктор регистрирует функцию с заданным объектом `stop_token`. Когда `stop_token` получает уведомление о поступившем запросе на прерывание потока, он вызывает для этого потока (в нашем примере — для главного потока) функцию, зарегистрированную через `stop_callback`. Для каждого объекта `stop_token` можно зарегистрировать через `stop_callback` любое количество функций. Порядок их выполнения не определен. Наша лямбда просто выводит на экран строку, подтверждающую вызов, но вместо этого вы могли бы освободить ресурсы перед завершением потока[2].

Анализ выходных данных

Выходные данные показывают, как спят и работают обе задачи. Когда функция `main` завершается, потоки `jthread`, выполняющие эти задачи, выходят из области видимости, поэтому их деструкторы вызывают функцию `request_stop` обоих объектов `jthread`, чтобы уведомить их задачи о том, что они должны завершиться. После строки `"MAIN (id: 16352) ENDS"` вы видите, когда каждый `jthread` получил вызов `request_stop`. Наш `stop_callback` отображает имя задачи и сообщение, включающее идентификатор потока, который велел задаче завершиться (в данном примере это главный поток)[3]. Обе задачи завершают работу.

17.10. Запуск задач с помощью шаблона функции std::async

На илл. 17.12 демонстрируется шаблон функции `std::async` — высокоуровневый способ запуска задачи в отдельном потоке. В этом примере мы реализуем потенциально длительную задачу: выяснить, является ли простым большое целое число, и найти его простые множители, если оно не простое.

Безопасность, шифрование и разложение огромных целых чисел на простые множители

Безопасность данных — одно из важнейших направлений программирования. Задача разложения чисел на простые множители[4, 5, 6] принципиально важна для

---

[1] «std::stop_callback». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/thread/stop_callback.

[2] Энтони Уильямс (Anthony Williams), «Concurrency in C++20 and Beyond».

[3] Если завершение запрошено до того, как сконструирован stop_callback, это будет идентификатор потока, конструирующего stop_callback (согласно https://en.cppreference.com/w/cpp/thread/stop_callback).

[4] «Prime Factorization», 2 июля 2020 г. Был доступен: 15 сентября 2023 г. https://www.cuemath.com/numbers/prime-factorization.

[5] Страйвер (Striver), «Prime Factors of a Big Number», 28 мая 2021 г. Был доступен: 15 сентября 2023 г. https://www.geeksforgeeks.org/prime-factors-big-number.

[6] Эхуд Шалит (Ehud Shalit), «Prime Numbers — Why Are They So Exciting?» 7 сентября 2018 г. Был доступен: 15 сентября 2023 г. https://kids.frontiersin.org/articles/10.3389/frym.2018.00040.

криптографического алгоритма RSA с открытым ключом, обычно применяемого для защиты данных, передаваемых через интернет[1, 2, 3]. Алгоритм RSA индустриального класса работает с огромными простыми числами, состоящими из сотен цифр. Для разложения такого числа на простые множители требуется невероятно много времени даже на самых мощных современных суперкомпьютерах, и именно по этой причине алгоритм RSA настолько безопасен. Криптографические алгоритмы с открытым ключом также используются в защите технологии блокчейн, на которой основаны такие криптовалюты, как Bitcoin[4]. Если вам интересно узнать, как работает RSA, прочитайте раздел «RSA Cryptography Appendix» на веб-странице нашей книги:

https://deitel.com/c-plus-plus-20-for-programmers

Обзор кода для работы с простыми числами

Задача этой программы выполняется функцией `getFactors` (строки 27–68), которая выясняет, является ли число простым[5], и если нет, находит его простые множители. Каждый поток в этой программе выполняется до тех пор, пока `getFactors` не вернет свой результат: кортеж `std::tuple`, содержащий имя задачи, проверяемое функцией `getFactors` число, переменную `bool`, указывающую, является ли число простым, и контейнер `vector` с множителями числа. Для того чтобы каждая из наших задач выполнялась хотя бы по несколько секунд, мы взяли два 19-значных числа, включая простое число с сайта «Prime Pages» Университета Теннесси-Мартин[6]. Там опубликованы результаты исследования простых чисел, в том числе их списки, отсортированные по количеству цифр.

Программа (илл. 17.12) состоит из следующих функций:

- Функция `id` (строки 13–17) преобразует уникальный идентификатор `std::thread::id` в `std::string`.
- Функция `getFactors` (строки 27–68) пытается найти множители целого числа.
- Функция `proveFactors` (строки 71–90) подтверждает, что результатом перемножения полученного набора простых множителей является соответствующее непростое число.

---

[1] «RSA (Cryptosystem)». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/RSA_(cryptosystem).

[2] «RSA Algorithm». Был доступен: 15 сентября 2023 г. https://simple.wikipedia.org/wiki/RSA_algorithm.

[3] К. Мориарти (K. Moriarty), Б. Калиски (B. Kaliski), Дж. Джонсон (J. Jonsson) и А. Руш (A. Rusch), «PKCS #1: RSA Cryptography Specifications Version 2.2», ноябрь 2016 г. Был доступен: 15 сентября 2023 г. https://tools.ietf.org/html/rfc8017.

[4] Сара Ротри (Sarah Rothrie), «How Blockchain Cryptography Is Fighting the Rise of Quantum Machines», 28 декабря 2018 г. Был доступен: 15 сентября 2023 г. https://coincentral.com/blockchain-cryptography-quantum-machines.

[5] «Primality Test». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Primality_test.

[6] «Index: Numbers». Был доступен: 15 сентября 2023 г. https://primes.utm.edu/curios/index.php.

- Функция `displayResults` (строки 93–117) выводит результаты выполнения задачи.
- Функция `main` (строки 119–133) запускает задачи с функцией `getFactors`, ожидает их завершения, а затем выводит результаты.

Мы разделили эту программу на части для удобства обсуждения. После исходного кода приведены примеры выходных данных. В строках 3–10 импортируются используемые в программе заголовки.

```
 1 // Fig. 17.12: async.cpp
 2 // Задачи поиска простых множителей, выполняемые в отдельных потоках
 3 #include <cmath>
 4 #include <fmt/format.h>
 5 #include <future> // Для std::async
 6 #include <iostream>
 7 #include <sstream>
 8 #include <string>
 9 #include <tuple>
10 #include <vector>
11
12 // Получаем идентификатор текущего потока в виде строки
13 std::string id() {
14    std::ostringstream out;
15    out << std::this_thread::get_id();
16    return out.str();
17 }
18
```

Илл. 17.12 | Задачи поиска простых множителей, запущенные с помощью std::async

Псевдонимы типов

В строках 20 и 24 определены псевдонимы типов, которые мы используем, чтобы упростить объявления типов:

- `Factors` (строка 20) — это псевдоним для вектора (`vector`), элементами которого являются пары чисел (`pair`). Каждая пара содержит простой множитель (`factor`) и количество операций деления на этот множитель без остатка (`count`), последовательно уменьшающих заданное число до минимума. Например, 8 можно разделить на 2 без остатка 3 раза подряд ($8 \div 2 \div 2 \div 2 = 1$), поэтому для числа 8 пара `factor` — `count` будет содержать числа 2 и 3 ($2^3 = 8$).
- `FactorResults` (строка 24) — это возвращаемый тип функции `getFactors`. Этот псевдоним представляет собой кортеж, содержащий: имя задачи (`std::string`); проверяемое число (`long long`); ответ на вопрос, является ли оно простым (`bool`); простые множители (`Factors`).

```
19 // Псевдоним типа для вектора, содержащего пары factor - count
20 using Factors = std::vector<std::pair<long long, int>>;
21
22 // Псевдоним типа для кортежа, содержащего 4 элемента:
23 // имя задачи, проверяемое число, простое ли оно, простые множители
24 using FactorResults = std::tuple<std::string, long long, bool, Factors>;
25
```

Функция getFactors для разложения целого числа на простые множители

Эта программа запускает отдельные потоки, каждый из которых вызывает функцию `getFactors` (строки 27–68), чтобы найти множители целого числа или выяснить, что оно простое. Функция получает имя задачи (для отображения в выходных данных) и проверяемое число, а возвращает объект типа `FactorResults`.

```
26 // Ищем простые множители
27 FactorResults getFactors(std::string name, long long number) {
28    std::cout << fmt::format(
29       "{}: Thread {} executing getFactors for {}\n", name, id(), number);
30
31    long long originalNumber{number}; // Копия, передаваемая в FactorResults
32    Factors factors; // Вектор для хранения пар factor - count
33
34    // Лямбда проверяет множитель и при успехе сохраняет пару factor - count
35    auto factorCount{
36       [&](int factor) {
37          int count{0}; // Счетчик выполненных операций деления без остатка
38
39          // Считаем операции деления без остатка на проверяемый множитель
40          while (number % factor == 0) {
41             ++count;
42             number /= factor;
43          }
44
45          // Добавляем в вектор пару factor - count
46          if (count > 0) {
47             factors.push_back({factor, count});
48          }
49       }
50    };
51
52    factorCount(2); // Считаем операции деления без остатка на 2
53
54    // Теперь число number стало нечетным; перебираем ряд нечетных i
55    for (int i{3}; i <= std::sqrt(number); i += 2) {
56       factorCount(i); // Считаем операции деления без остатка на i
57    }
58
59    // Добавляем в вектор последнюю пару factor - count
60    if (number > 2) {
```

```
61        factors.push_back({number, 1});
62    }
63
64    bool isPrime{factors.size() == 1 && get<int>(factors[0]) == 1};
65
66    // Инициализируем объект FactorResults полученными данными
67    return {name, originalNumber, isPrime, factors};
68 }
69
```

Функция работает таким образом:

- В строках 28–29 выводится на экран имя выполняемой задачи вместе с идентификатором потока, в котором выполняется функция `getFactors`.
- Если число `number` имеет простые множители, наш алгоритм его изменит, поэтому в строке 31 `number` копируется для последующей передачи в `FactorResults`.
- В строке 32 определен объект `Factors`: вектор, содержащий пары `factor` — `count`.
- В строках 35–50 определена лямбда `factorCount`, которая захватывает по ссылке число `number` и уменьшает его последовательными операциями деления без остатка на `factor` (аргумент лямбды), подсчитывая, сколько таких операций будет выполнено. Если `number` возможно разделить без остатка на `factor` (строка 40), то в строке 41 `count` увеличивается на `1`, затем в строке 42 `number` делится на `factor` и цикл повторяется. После этого, если `count` больше `0`, пара `factor` — `count` добавляется в вектор `factors` (строка 47).
- В строке 52 вызывается функция `factorCount` для множителя `2`. Затем в строках 55–57 эта функция многократно вызывается для нечетных чисел от `3` и больше. Это происходит, пока `i` меньше или равно квадратному корню `number`.
- В строках 60–62 выполняется проверка, по-прежнему ли `number` больше `2`. Если да, то это последний множитель, который надо добавить в `factors`.
- Если исходное число `number` простое, то `factors` будет содержать только само это число, а значение `count` будет равно `1`. Инструкция в строке 64 проверяет это и присваивает соответствующее значение переменной `bool` с именем `isPrime`.
- Наконец, в строке 67 инициализируется кортеж `FactorResults`, который возвращает функция `getFactors`. Кортеж содержит: имя задачи (`name`); исходное число (`originalNumber`); ответ на вопрос, является ли оно простым (`isPrime`); и найденные множители (`factors`).

Функция proveFactors для проверки найденных множителей

В строках 71–90 определена функция `proveFactors`, которая подтверждает, что заданное число является результатом произведения его простых множителей:

- В сроке 72 переменная `proof` инициализируется значением `1`.
- В строках 77–79 для каждой пары `factor` – `count` из вектора `factors` (строка 76) выполняется `count` последовательных операций умножения `proof` на `factor`.
- В строках 83–89 проверяется равенство чисел `number` и `proof`, после чего выводится соответствующее сообщение.

```
70 // Перемножаем множители и подтверждаем, что результат равен заданному числу
71 void proveFactors(long long number, const Factors& factors) {
72    long long proof{1};
73
74    // Извлекаем из вектора каждую пару factor - count
75    // и выполняем с ней count операций умножения proof на factor
76    for (const auto& [factor, count] : factors) {
77       for (int i{0}; i < count; ++i) {
78          proof *= factor;
79       }
80    }
81
82    // Подтверждаем равенство чисел proof и number
83    if (proof == number) {
84       std::cout << fmt::format(
85          "\nProduct of factors matches original value ({})\n", proof);
86    }
87    else {
88       std::cout << fmt::format("\n{} != {}\n", proof, number);
89    }
90 }
91
```

Функция displayResults для отображения результатов

В строках 93–117 определена функция `displayResults`, которая вызывается из `main` для вывода на экран данных из каждого кортежа `FactorResults`, полученного от задач этой программы:

- В строке 96 кортеж распаковывается в переменные `name` (`std::string`), `number` (`long long`), `isPrime` (`bool`) и `factors` (`Factors`). Типы этих переменных автоматически определяются по типу кортежа (строка 24).
- В строке 98 выводится на экран имя задачи.
- Если проверяемое число простое (строка 101), в строке 102 выводится на экран сообщение. Если это число непростое, в строках 105–110 выводятся на экран его простые множители с соответствующими показателями степени (`count`).
- Наконец, если проверяемое число непростое, в строке 115 вызывается функция `proveFactors`, подтверждающая, что результатом перемножения полученных чисел является проверяемое число.

```
92 // Выводим на экран результаты вычислений
93 void displayResults(const FactorResults& results) {
94    // Распаковываем results в переменные name (std::string),
95    // number (long long), isPrime (bool) и factors (Factors)
96    const auto& [name, number, isPrime, factors] {results};
97
98    std::cout << fmt::format("\n{} results:\n", name);
99
100    // Сообщаем, оказалось ли число простым
101    if (isPrime) {
102       std::cout << fmt::format("{} is prime\n", number);
103    }
104    else { // Показываем простые множители
105       std::cout << fmt::format("{}'s prime factors:\n\n", number);
106       std::cout << fmt::format("{:<12}{:<8}\n", "Factor", "Count");
107
108       for (const auto& [factor, count] : factors) {
109          std::cout << fmt::format("{:<12}{:<8}\n", factor, count);
110       }
111    }
112
113    // Если число непростое, вызываем функцию, которая проверит вычисления
114    if (!isPrime) {
115       proveFactors(number, factors);
116    }
117 }
118
```

Создание и выполнение задачи: шаблон функции std::async

Заголовок <future> (строка 5) содержит функции, которые позволяют вам выполнять асинхронные задачи и получать результаты этих задач по завершении их выполнения.

```
119 int main() {
120    std::cout << "MAIN LAUNCHING TASKS\n";
121    auto future1{std::async(std::launch::async,
122       getFactors, "Task 1", 101666900611668299З)}; // Не простое число
123    auto future2{std::async(std::launch::async,
124       getFactors, "Task 2", 10000000000000000003)}; // Простое число
125
126    std::cout << "\nWAITING FOR TASK RESULTS\n";
127
128    // Ждем результатов от каждой задачи, затем выводим их на экран
129    displayResults(future1.get());
130    displayResults(future2.get());
131
132    std::cout << "\nMAIN ENDS\n";
133 }
```

```
MAIN LAUNCHING TASKS
WAITING FOR TASK RESULTS
```

```
Task 1: Thread 5032 executing getFactors for 1016669006116682993
Task 2: Thread 2952 executing getFactors for 1000000000000000003
Task 1 results:
1016669006116682993's prime factors:
Factor Count
1000000007 1
1016668999 1
Product of factors matches original value (1016669006116682993)
Task 2 results:
1000000000000000003 is prime
MAIN ENDS
```

В строках 121–122 и 123–124 с помощью шаблона функции `std::async`[1] создаются два потока, выполняющих `getFactors` асинхронно. Функция `std::async` имеет две версии. Мы вызываем ту версию, которая принимает три аргумента:

- Политика запуска (из перечисления `std::launch`) — это или `std::launch::async`, или `std::launch::deferred`, или обе эти политики, разделенные оператором побитового ИЛИ (`|`). С политикой `std::launch::async` функция, указанная во втором аргументе, будет выполняться в отдельном потоке. С политикой `std::launch::deferred` она будет выполняться в том же потоке. С комбинацией двух политик система самостоятельно выберет, как лучше выполнять функцию — асинхронно или синхронно.
- Задача для выполнения — это функция, функциональный объект или лямбда.
- Любые последующие аргументы передаются через `std::async` в функцию задачи. Мы передали строковое имя задачи и число `long long`, которое надо разложить на простые множители.

Другая версия функции `std::async` не получает аргумент политики запуска, выбирая политику самостоятельно.

Если `std::async` получает политику запуска `std::launch::async`, но не может создать поток, она выдает исключение `std::system_error`. Если `std::async` успешно создает поток, функция задачи начинает выполняться в новом потоке.

Объекты std::future и std::promise для связи между потоками

Функция `std::async` возвращает объект шаблона класса `std::future` (C++11), который обеспечивает связь между потоком, вызывающим `async`, и задачей, которую выполняет `async`. В C++ Core Guidelines рекомендуется использовать `future`, чтобы возвращать результат асинхронной задачи[2].

Функция `async` использует «под капотом» объект `std::promise`, из которого получает возвращенный объект `future`. Когда задача завершается, `async` сохраняет ее

[1] «std::async». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/thread/async.

[2] C++ Core Guidelines, «CP.60: Use a future to Return a Value from a Concurrent Task». Был доступен: 15 сентября 2023 г. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconc-future.

результат в объекте promise. Вызывающая сторона async использует future для доступа к результату в promise — в нашем случае это объект FactorResults. Вам не нужно работать с объектом promise напрямую.

Чтобы гарантировать, что программа не завершится до завершения задач, и получить результаты каждой задачи, в строках 129–130 вызывается функция get каждого объекта future. Если задача все еще выполняется, get блокирует вызывающий поток, который ждет завершения задачи; в противном случае get немедленно возвращает результат задачи. Возвращаемый тип функции get — это то, что возвращает функция задачи. Когда результаты доступны, в строках 129–130 они передаются в функцию displayResults. В выходных данных программы показаны уникальные идентификаторы потоков, подтверждающие, что обе задачи getFactors выполнялись в отдельных потоках.

Если выполняемая через async задача выдает исключение, объект future содержит исключение, а не результат задачи, а вызов get выдаст исключение повторно. Если доступ к результату асинхронной задачи нужен сразу нескольким потокам, можно использовать объекты std::shared_future[1].

Класс std::packaged_task как альтернатива std::async

Другой способ выполнения асинхронных задач — класс std::packaged_task[2]. Основные отличия этого способа от рассмотренного ранее (с помощью async) заключаются в следующем:

- Вы должны вызвать функцию get_future класса std::packaged_task, чтобы получить соответствующий объект future для получения результатов задачи впоследствии.
- Функция packaged_task выполняется в потоке, который вызывает функцию operator() задачи.
- Чтобы выполнить packaged_task в отдельном потоке, вы создаете поток и инициализируете его с помощью std::move(*вашОбъектPackagedTask*). Когда поток завершится, вы можете вызвать функцию get объекта future задачи, чтобы получить результат.

17.11. Однократная потокобезопасная инициализация

Иногда переменная должна быть инициализирована ровно один раз, даже когда инструкцию ее инициализации пытаются выполнить конкурентные потоки. До C++11 часто использовался метод, называемый блокировкой с двойной провер-

[1] «std::shared_future». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/thread/shared_future.

[2] «std::packaged_task». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/thread/packaged_task.

кой, но он не гарантировал потокобезопасность во всех компиляторах и на всех платформах[1]. Здесь мы обсудим два введенных в C++11 метода однократной потокобезопасной инициализации переменных.

Статические локальные переменные

Когда в функции определена статическая локальная переменная и параллельные потоки выполняют эту функцию, в соответствии со стандартом C++ инициализация переменной «выполняется в первый раз, когда поток управления проходит через ее объявление»[2,3]. Таким образом, первый поток, выполняющий объявление статической локальной переменной, ее инициализирует. Все остальные потоки, пытающиеся выполнить объявление статической переменной, блокируются до тех пор, пока первый поток не завершит инициализацию переменной. Затем другие потоки, вызывающие функцию, пропускают это объявление и используют уже инициализированную статическую локальную переменную.

Объект once_flag и функция call_once

Объект `std::once_flag`[4] и функцию `std::call_once`[5] (заголовок `<mutex>`), введенные в C++11, вы можете использовать, когда надо гарантировать, что конкурентные потоки выполнят функцию, лямбду или объект функции ровно один раз. Сначала объявите объект `once_flag`:

```
std::once_flag myFlag;
```

Затем используйте `call_once` для вызова вашей функции:

```
std::call_once(myFlag, myFunction, аргументы);
```

Первый поток, выполнивший эту инструкцию, вызовет функцию `myFunction`, передав ей указанные вами аргументы (если они есть). Когда эту же инструкцию будет выполнять другой поток, функция `call_once` немедленно вернет управление, не вызывая `myFunction`. Распространенный пример применения объекта `call_once` приведен в видеоролике Артура О'Двайра «Back to Basics: Concurrency»[6].

---

[1] Скотт Майерс (Scott Meyers) и Андрей Александреску (Andrei Alexandrescu), «C++ and the Perils of Double-Checked Locking», сентябрь 2004 г. Был доступен: 15 сентября 2023 г. https://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf.

[2] Стандарт C++, «8.8. Declaration Statement». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/stmt.dcl#4.

[3] «Storage Class Specifiers — static Local Varables». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/language/storage_duration#Static_local_variables.

[4] «std::once_flag». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/thread/once_flag.

[5] «std::call_once». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/thread/call_once.

[6] Артур О'Двайр (Arthur O'Dwyer), «Back to Basics: Concurrency», 6 октября 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=F6Ipn7gCOsY.

17.12. Основы атомарных типов

В примерах кода из разделов 17.6–17.7 исключительный доступ к общим изменяемым данным обеспечивался с помощью примитивов синхронизации: мьютексов `std::mutex`, блокировок и условных переменных `std::condition_variable`. В C++11 появились **атомарные типы (atomic types)**[1] (заголовок `<atomic>`[2]), позволяющие выполнять операции, которые нельзя разделить на более мелкие шаги. Благодаря неделимости атомарных операций, потоки могут совместно использовать изменяемые данные без явной синхронизации и блокировки.

Строго говоря, атомарные типы относятся к более высокому уровню, чем мьютексы, блокировки и условные переменные. Однако атомарные типы и их операции считаются в первую очередь низкоуровневыми средствами, которые помогают разработчикам библиотек реализовать высокоуровневые средства многопоточности. Например, стандартные библиотеки Visual C++, GNU C++ и Clang C++ используют атомарные типы «под капотом» для реализации введенных в C++20 примитивов координации потоков `std::latch`, `std::barrier` и `std::semaphore` (разделы 17.13 и 17.14)[3,4,5].

Мы приведем простые примеры кода с атомарными типами. Для большинства разработчиков удобнее примитивы более высокого уровня, которые помогают им быстро создавать корректные, надежные и эффективные многопоточные приложения. В статье «A Concurrency Cost Hierarchy» на GitHub подробно обсуждается эффективность атомарных типов в сравнении с другими технологиями[6].

Изменение переменных int, std::atomic<int> и std::atomic_ref<int> из двух конкурентных потоков

На илл. 17.13 показан пример с двумя конкурентными потоками, изменяющими:

- незащищенную переменную типа `int`;
- объект типа `std::atomic<int>`;
- объект шаблона класса `std::atomic_ref<int>`.

---

[1] Ханс-Дж. Бом (Hans-J. Boehm) и Лоуренс Кроул (Lawrence Crowl), «C++ Atomic Types and Operations», 3 октября 2007 г. Был доступен: 15 сентября 2023 г. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2427.html.

[2] «Atomic Operations Library». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/atomic.

[3] «Microsoft's C++ Standard Library». Был доступен: 15 сентября 2023 г. https://github.com/microsoft/STL.

[4] «libstdc++». Был доступен: 15 сентября 2023 г. https://gcc.gnu.org/onlinedocs/libstdc++/latestdoxygen/files.html.

[5] Документация на libc++. Был доступен: 15 сентября 2023 г. https://github.com/llvm/llvm-project/tree/main/libcxx.

[6] Трэвис Доунс (Travis Downs), «A Concurrency Cost Hierarchy», 6 июля 2020 г. Был доступен: 15 сентября 2023 г. https://travisdowns.github.io/blog/2020/07/06/concurrency-costs.html.

Введенный в C++20 шаблон класса `std::atomic_ref` позволяет выполнять атомарные операции с переменными по ссылке (в нашей программе это переменная типа `int`). Сначала мы покажем в действии `std::atomic<int>` и `std::atomic_ref<int>`, а затем расскажем немного больше об атомарных типах.

```
// Fig. 17.13: atomic.cpp
// Изменение целочисленных переменных из конкурентных потоков
// с помощью атомарных типов и без них
#include <atomic>
#include <fmt/format.h>
#include <iostream>
#include <thread>

int main() {
   int count1{0};
   std::atomic<int> atomicCount{0};
   int count2{0};
   std::atomic_ref<int> atomicRefCount{count2};

   {
      std::cout << "Two concurrent threads incrementing int count1, "
         << "atomicCount and atomicRefCount\n\n";

      // Лямбда, инкрементирующая переменные
      auto incrementer{
         [&]() {
            for (int i{0}; i < 1000; ++i) {
               ++count1; // Нет синхронизации
               ++atomicCount; // Операция выполняется как атомарная
               ++atomicRefCount; // Операция выполняется как атомарная
               std::this_thread::yield(); // Забираем у потока процессор
            }
         }
      };

      std::jthread t1{incrementer};
      std::jthread t2{incrementer};
   }

   std::cout << fmt::format("Final count1: {}\n", count1);
   std::cout << fmt::format("Final atomicCount: {}\n", atomicCount);
   std::cout << fmt::format("Final count2: {}\n", count2);
}
```

```
Two concurrent threads incrementing int count1, atomicCount and
atomicRefCount

Final count1: 2000
Final atomicCount: 2000
Final count2: 2000
```

Окончание ⇨

```
Two concurrent threads incrementing int count1, atomicCount and
atomicRefCount

Final count1: 1554
Final atomicCount: 2000
Final count2: 2000
```

Илл. 17.13 | Изменение целочисленных переменных из конкурентных потоков с помощью атомарных типов и без них

Определение переменных

В строках 10–13 определены:

- переменная типа `int` с именем `count1`;
- переменная типа `std::atomic<int>` с именем `atomicCount`;
- переменная типа `int` с именем `count2`;
- переменная типа `std::atomic_ref<int>` с именем `atomicRefCount`.

Первые три переменные инициализируются значением `0`, а `atomicRefCount` инициализируется ссылкой на переменную `count2`. Мы будем использовать `atomicRefCount`, чтобы показать, что можно косвенно выполнять с переменной `count2` атомарные операции по ссылке.

Изменение переменных

В строках 15–33 используются два несинхронизированных конкурентных потока, каждый из которых `1000` раз выполняет операцию инкремента трех переменных. Если все работает правильно, то после завершения потоков каждая переменная должна иметь значение `2000`. В строках 20–29 определена лямбда `incrementer`, инкрементирующая переменные `count1` (строка 23), `atomicCount` (строка 24) и `atomicRefCount` (строка 25) оператором `++`. Это один из тех немногих операторов, которые можно применять к атомарным объектам. Когда один поток выполняет операцию `++` с атомарным объектом, гарантируется, что эта операция завершится до того, как другой поток сможет прочитать или изменить значение этого атомарного объекта. Для обычной (не атомарной) переменной типа `int` это не гарантируется. Полный список атомарных операций приведен здесь:

https://en.cppreference.com/w/cpp/atomic/atomic

На мощном современном процессоре поток может выполнить весь цикл очень быстро, поэтому несколько потоков, увеличивающих переменную `int` без синхронизации, иногда могут работать корректно, как вы видите в первом примере выходных данных этой программы. По этой причине в строке 26 вызывается функция `yield` из пространства имен `std::this_thread`, чтобы заставить выполняющийся поток отказаться от процессора после каждой итерации цикла. Раньше мы переводили потоки в сон, а теперь вызываем `yield` с этой же целью: продемон-

стрировать невозможность предсказания относительной скорости асинхронных конкурентных потоков.

В строках 31–32 создаются объекты `std::jthread`, каждый из которых изменяет переменные. Затем блок завершается и объекты `jthread` выходят из области видимости, автоматически присоединяя потоки. Наконец, в строках 35–37 выводятся на экран значения `count1`, `atomicCount` и `count2`. Переменная `count2` была увеличена косвенно через `atomicRefCount`.

Анализ выходных данных

В строках 20–29 переменные `count1`, `atomicCount` и `count2` (косвенно через `atomicRefCount`) должны быть инкрементированы `2000` раз. Однако примеры выходных данных показывают, что операции с переменной `count` типа `int`, выполняемые из конкурентных потоков без синхронизации, могут каждый раз давать разные результаты, включая кажущийся правильным результат в первом примере. Но аналогичные операции с переменной `atomicCount`, а также операции с переменной `count2`, выполняемые косвенно через `atomicRefCount`, гарантированно приводят к правильному результату `2000`, несмотря на отсутствие синхронизации между конкурентными потоками.

Подробнее об атомарных типах[1, 2]

Предопределенные специализации шаблона класса `std::atomic` и соответствующие псевдонимы типов существуют для типа `bool` и для каждого целочисленного типа. Кроме того, вы можете специализировать `std::atomic` для:

- типов указателей;
- типов с плавающей точкой;
- тривиально копируемых типов[3] (это означает, что компилятор предоставляет для них копирующие и перемещающие конструкторы, операторы копирующего и перемещающего присваивания, а также деструкторы; виртуальные функции отсутствуют).

Специализации для целочисленных типов, для типов указателей и, начиная с C++20, для типов с плавающей точкой поддерживают операции сложения и вычитания с атомарными элементами данных. Целочисленные специализации также поддерживают атомарные побитовые операции `&=` (И), `|=` (ИЛИ) и `^=` (исключающее ИЛИ).

---

[1] Стандарт C++, «Atomic Operations Library». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/atomics.

[2] «std::atomic». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/atomic/atomic.

[3] Стандарт C++, «Properties of Classes». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/class.prop.

Атомарные умные указатели (C++20) и атомарные указатели[1, 2]

В C++20 введены специализации атомарных умных указателей для `std::atomic<shared_ptr<T>>` и `std::atomic<weak_ptr<T>>`, позволяющие выполнять атомарные операции с умными указателями этих типов.

Для всех атомарных указателей атомарными являются только операции, выполняемые непосредственно с указателями (а не с объектами, на которые они указывают), например:

- перенаправление указателя с одного объекта на другой;
- инкрементирование указателя;
- декрементирование указателя.

Атомарные указатели можно использовать для реализации потокобезопасных связных структур данных.

Шаблон класса std::atomic_ref (C++20)

Как и `std::atomic`, шаблон класса `std::atomic_ref` может быть специализирован для любого примитивного типа, для типа указателя или тривиально копируемого типа. Однако объект `std::atomic_ref` инициализируется ссылкой, а не значением. Вы можете использовать `atomic_ref` для выполнения атомарных операций над объектом, на который ссылаетесь. Этот объект должен существовать в программе дольше, чем ссылающийся на него `atomic_ref`, иначе возникнет висячая ссылка или висячий указатель[3, 4]. Несколько `atomic_ref` могут ссылаться на один и тот же объект.

17.13. Координация потоков с помощью защелок и барьеров (C++20)

До сих пор мы координировали потоки с помощью мьютексов `std::mutex`, условных переменных `std::condition_variable` и блокировок. C++20 предоставляет для координации потоков более высокоуровневые типы `std::latch` (защелки) и `std::barrier` (барьеры)[5], которые не требуют явного использования мьютексов, условных переменных и блокировок[6].

---

[1] Стандарт C++, «Partial Specializations for Smart Pointers». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/util.smartptr.atomic.

[2] «std::atomic». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/atomic/atomic.

[3] Стандарт C++, «31.7. Class Template atomic_ref». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/atomics.ref.generic.

[4] «std::atomic_ref». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/atomic/atomic_ref.

[5] Стандарт C++, «Thread Support Library — Coordination Types». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/thread.coord.

[6] Брайс Адельштейн Лельбах (Bryce Adelstein Lelbach), «The C++20 Synchronization Library», 24 октября 2019 г. https://www.youtube.com/watch?v=Zcqwb3CWqs4.

17.13.1. Защелки класса std::latch (C++20)

Защелка (latch) — это объект `std::latch` (заголовок `<latch>`), представляющий собой одноразовый шлюз в коде, который изначально закрыт. Когда несколько потоков по очереди сообщат защелке о своем завершении, защелка открывается и остается постоянно открытой[1]. Защелка служит в качестве одноразовой точки синхронизации, которая заставляет потоки ждать, пока определенное количество потоков не достигнет этой точки. Защелки проще применять для координации потоков, чем мьютексы и условные переменные.

Рассмотрим параллельный алгоритм сортировки, который:

- запускает несколько рабочих (выполняющих задачи) потоков, каждый из которых сортирует свою часть большого массива;
- ожидает завершения этих потоков;
- объединяет отсортированные части массива в окончательный отсортированный массив.

Предположим, что алгоритм использует два рабочих потока, каждый из которых сортирует половину массива. Алгоритм может использовать объект `std::latch`, чтобы дождаться их завершения:

- Сначала алгоритм создает защелку `std::latch` с ненулевым значением счетчика. В данном случае надо дождаться завершения двух рабочих потоков, поэтому счетчик инициализируется значением `2`.
- Оба рабочих потока связаны с одной и той же защелкой.
- После запуска потоков алгоритм блокируется этой защелкой и ожидает ее открытия, которое позволит ему продолжать выполнение.
- Когда любой из рабочих потоков достигает защелки, он уменьшает ее счетчик на `1` (отправляет защелке сигнал), сообщая тем самым, что он закончил сортировку своей части массива.
- Когда счетчик защелки становится равным `0`, защелка открывается, снимая блокировку потока алгоритма, чтобы он объединил результаты своих рабочих потоков.

Попытка ожидания, когда шлюз постоянно открыт

Любое количество потоков может ждать открытия конкретной защелки. Но как только она открывается, все потоки, ожидавшие ее открытия, просто проходят через этот шлюз и продолжают выполнение.

Пример кода с защелкой

В показанной на илл. 17.14 программе лямбда с именем `task` (строки 22–34) предназначена для выполнения задачи, которая должна завершиться до того, как

[1] Стандарт C++, «Thread Support Library — Coordination Types — Latches». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/thread.latch.

функция `main` получит разрешение продолжить выполнение. Как мы уже делали на протяжении этой главы, мы имитируем задачу переводом потока в сон. Программа работает так:

- В строке 19 создается защелка: объект `std::latch` с именем `mainLatch` и начальным значением счетчика `3`. Никакие потоки, ожидающие открытия защелки `mainLatch`, не могут продолжать выполнение, пока ее не достигнут три потока.
- В строках 40–46 запускаются три потока `jthread`, каждый из которых выполняет лямбду `task`.
- После того как `main` запустит потоки `jthread`, в строке 49 вызывается функция `wait` объекта `mainLatch`. Если счетчик `mainLatch` больше `0`, функция `main` ждет в строке 49, пока счетчик не уменьшится до `0`.
- Когда выполняемая задача в каждом потоке `jthread` достигает строки 32, она вызывает функцию `count_down` объекта `mainLatch`, чтобы подать защелке сигнал и декрементировать ее счетчик. Хотя в этой программе задача завершается, она могла бы продолжать работать. Главное, чтобы та часть задачи, которую ожидает `main`, была завершена каждым потоком до того, как он подаст сигнал защелке. В выходных данных обратите внимание, что потоки подают защелке сигналы в случайном порядке.
- Когда счетчик защелки `mainLatch` уменьшается до `0`, все потоки, ожидающие ее открытия, разблокируются и продолжают выполнение. В этой программе `main` продолжает выполнение в строке 51.
- Чтобы показать, что защелка является одноразовым шлюзом, в строке 53 снова вызывается функция `wait` объекта `mainLatch`. Поскольку защелка уже открыта, `main` просто продолжает выполнение в строке 54, выводит еще одно сообщение и завершает работу.

```
 1 // Fig. 17.14: LatchDemo.cpp
 2 // Координация потоков с помощью защелки - объекта std::latch
 3 #include <chrono>
 4 #include <fmt/format.h>
 5 #include <iostream>
 6 #include <latch>
 7 #include <random>
 8 #include <string_view>
 9 #include <thread>
10 #include <vector>
11
12 int main() {
13    // Инициализируем генератор случайных чисел
14    std::random_device rd;
15    std::default_random_engine engine{rd()};
16    std::uniform_int_distribution ints{2000, 3000};
17
18    // Три потока должны подать сигнал этой защелке для продолжения main
19    std::latch mainLatch{3};
20
```

```
    // Лямбда с задачей, выполняемая тремя потоками
    auto task{
       [&](std::string_view name, std::chrono::milliseconds workTime) {
          std::cout << fmt::format("Proceeding with {} work for {} ms.\n",
             name, workTime.count());

          // Переводим поток в сон для имитации работы
          std::this_thread::sleep_for(workTime);

          // Сообщаем, что задача достигла защелки mainLatch
          std::cout << fmt::format("{} done; signals mainLatch.\n", name);
          mainLatch.count_down();
       }
    };

    std::vector<std::jthread> threads; // Для хранения объектов потоков
    std::cout << "Main starting three jthreads.\n";

    // Запускаем три потока
    for (int i{1}; i < 4; ++i) {
       // Создаем jthread, который вызывает лямбду с задачей,
       // передаем имя задачи и время работы
       threads.push_back(std::jthread{task,
          fmt::format("Task {}", i),
          std::chrono::milliseconds{ints(engine)}});
    }

    std::cout << "\nMain waiting for jthreads to reach the latch.\n\n";
    mainLatch.wait();

    std::cout << "\nAll jthreads reached the latch. Main working.\n";
    std::cout << "Showing that mainLatch is permanently open.\n";
    mainLatch.wait(); // Защелка уже открыта
    std::cout << "mainLatch is already open. Main continues.\n";
55 }
```

```
Main starting three jthreads.

Main waiting for jthreads to reach the latch.

Proceeding with Task 3 work for 2648 ms.
Proceeding with Task 2 work for 2705 ms.
Proceeding with Task 1 work for 2024 ms.
Task 1 done; signals mainLatch.
Task 3 done; signals mainLatch.
Task 2 done; signals mainLatch.

All jthreads reached the latch. Main working.
Showing that mainLatch is permanently open.
mainLatch is already open. Main continues.

Main starting three jthreads.

Main waiting for jthreads to reach the latch.
```

Окончание ⇨

```
Proceeding with Task 2 work for 2571 ms.
Proceeding with Task 1 work for 2462 ms.
Proceeding with Task 3 work for 2248 ms.
Task 3 done; signals mainLatch.
Task 1 done; signals mainLatch.
Task 2 done; signals mainLatch.

All jthreads reached the latch. Main working.
Showing that mainLatch is permanently open.
mainLatch is already open. Main continues.
```

Илл. 17.14 | Координация потоков с помощью защелки — объекта std::latch

17.13.2. Барьеры класса std::barrier (C++20)

Представьте себе участок покраски на автоматическом конвейере сборки автомобилей. Часто несколько роботов, управляемых компьютером, работают на участке покраски совместно. Предположим, что автомобили движутся по конвейеру, а действия двух роботов управляются двумя потоками программы. Как только покраска очередного автомобиля заканчивается, надо сбросить текущие состояния потоков, продвинуть конвейер и снова выполнить ту же работу для следующего автомобиля. Для реализации этого сценария идеально подходит **барьер (barrier)** — объект `std::barrier`[1] (заголовок `<barrier>`), похожий на многоразовую защелку. Обычно барьер используется для задач, повторяющихся в цикле:

- Каждый поток работает, затем достигает барьера и ждет, пока барьер откроется.
- Когда заданное количество потоков достигает барьера, выполняется опциональная функция завершения.
- Барьер сбрасывает свой счетчик и открывается для потоков. Теперь они могут продолжить выполнение и повторить весь процесс.

Используя барьер (илл. 17.15), мы смоделируем этап покраски ряда автомобилей на конвейере. Мы также будем использовать `stop_source` и `stop_token` вручную, чтобы координировать отмену потоков при остановке конвейера. И мы будем снова использовать `sleep` для имитации работы. Обратите внимание, что «роботы» могут заканчивать свою работу с автомобилем в случайном порядке.

```
1 // Fig. 17.15: BarrierDemo.cpp
2 // Координация потоков с помощью барьера - объекта std::barrier
3 #include <barrier>
4 #include <chrono>
5 #include <fmt/format.h>
6 #include <iostream>
7 #include <random>
8 #include <string_view>
```

[1] Стандарт C++, «Thread Support Library — Coordination Types — Barriers». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/thread.barrier.

```
#include <thread>

int main() {
   // Имитируем перемещение автомобиля на участок покраски
   auto moveCarIntoPosition{
      []() {
         std::cout << "Moving next car into painting position.\n";
         std::this_thread::sleep_for(std::chrono::seconds(1));
         std::cout << "Car ready for painting.\n\n";
      }
   };

   int carsToPaint{3};

   // stop_source используется для уведомления роботов об остановке конвейера
   std::stop_source assemblyLineStopSource;

   // stop_token сообщает лямбде paintingRobotTask о необходимости завершения
   std::stop_token stopToken{assemblyLineStopSource.get_token()};

   // Конвейер ждет, пока два покрасочных робота достигнут этого барьера
   std::barrier paintingDone{2,
      [&]() noexcept { // Лямбда, вызываемая при завершении работы роботов
         static int count{0}; // Количество покрашенных автомобилей
         std::cout << "Painting robots completed tasks\n\n";

         // Проверяем, не пора ли остановить конвейер
         if (++count == carsToPaint) {
            std::cout << "Shutting down assembly line\n\n";
            assemblyLineStopSource.request_stop();
         }
         else {
            moveCarIntoPosition();
         }
      }
   };

   // Лямбда, имитирующая работу по покраске автомобиля
   auto paintingRobotTask{
      [&](std::string_view name) {
         // Инициализируем генератор случайных чисел
         std::random_device rd;
         std::default_random_engine engine{rd()};
         std::uniform_int_distribution ints{2500, 5000};

         // Проверяем, остановлен ли конвейер,
         // и если нет, выполняем покраску
         while (!stopToken.stop_requested()) {
            auto workTime{std::chrono::milliseconds{ints(engine)}};

            std::cout << fmt::format("{} painting for {} ms\n",
               name, workTime.count());
            std::this_thread::sleep_for(workTime); // Имитируем работу
```

Окончание ⇨

```
62
63              // Сообщаем, что поток проснулся и достиг барьера
64              std::cout << fmt::format(
65                  "{} done painting. Waiting for next car.\n", name);
66
67              // Уменьшаем счетчик барьера paintingDone
68              // и ждем, когда этой точки достигнут другие покрасочные роботы
69              paintingDone.arrive_and_wait();
70          }
71
72          std::cout << fmt::format("{} shut down.\n", name);
73      }
74  };
75
76  moveCarIntoPosition(); // Перемещаем первый автомобиль на участок покраски
77
78  // Запускаем потоки с задачами покрасочных роботов
79  std::cout << "Starting robots.\n\n";
80  std::jthread leftSideRobot{paintingRobotTask, "Left side robot"};
81  std::jthread rightSideRobot{paintingRobotTask, "Right side robot"};
82 }
```

```
Moving next car into painting position.
Car ready for painting.

Starting robots.

Left side robot painting for 4564 ms
Right side robot painting for 2758 ms
Right side robot done painting. Waiting for next car.
Left side robot done painting. Waiting for next car.
Painting robots completed tasks

Moving next car into painting position.
Car ready for painting.

Left side robot painting for 4114 ms
Right side robot painting for 2860 ms
Right side robot done painting. Waiting for next car.
Left side robot done painting. Waiting for next car.
Painting robots completed tasks

Moving next car into painting position.
Car ready for painting.

Right side robot painting for 4730 ms
Left side robot painting for 3794 ms
Left side robot done painting. Waiting for next car.
Right side robot done painting. Waiting for next car.
Painting robots completed tasks

Shutting down assembly line

Right side robot shut down.
Left side robot shut down.
```

Илл. 17.15 | Координация потоков с помощью барьера — объекта std::barrier

Поток с функцией `main` управляет конвейером. Программа работает таким образом:

- В строках 13–19 определена лямбда, имитирующая перемещение очередного автомобиля на участок покраски.
- В строке 21 определена переменная `carsToPaint` — количество автомобилей, которые надо покрасить.
- Мы используем кооперативное прерывание (раздел 17.9), чтобы завершить потоки роботов, когда конвейер останавливается. Деструктор каждого объекта `jthread` перед присоединением потока вызывает функцию `request_stop` своего объекта `stop_source`. Затем задача потока `jthread` может проверить свой параметр `stop_token`, чтобы узнать, надо ли завершить выполнение. В этой симуляции мы не хотим, чтобы потоки роботов завершались при выходе объектов `jthread` из области видимости в конце функции `main`, поэтому мы прерываем операции вручную. Объект `stop_source` (строка 24) координирует прерывание с потоками роботов. Когда три автомобиля будут покрашены, мы вызовем функцию `request_stop` этого объекта `stop_source`, чтобы уведомить потоки роботов о том, что конвейер останавливается.
- В строке 27 мы получаем `stop_token` объекта `stop_source`. Перед покраской каждый поток робота вызывает функцию `stop_requested` этого `stop_token`, чтобы проверить, останавливается ли конвейер.
- В строках 30–44 определен барьер `paintingDone` с начальным значением счетчика `2` (для двух потоков) и с функцией завершения (строки 31–43), которая вызывается, когда счетчик уменьшится до `0`. Статическая локальная переменная `count` отслеживает количество покрашенных автомобилей. Если значение `count` равно `carsToPaint`, в строке 38 вызывается функция `request_stop` объекта `stop_source`, чтобы сообщить об остановке конвейера. В противном случае в строке 41 выполняется перемещение следующего автомобиля на участок покраски.
- В строках 47–74 определена лямбда `paintingRobotTask`. Строки 56–70 выполняются до тех пор, пока в строке 56 не будет определено, что задача должна быть остановлена, поскольку конвейер останавливается. В строках 57–65 имитируется работа по покраске. Когда покраска заканчивается, вызвавший задачу поток `jthread` достигает барьера `paintingDone` и вызывает его функцию `arrive_and_wait` (строка 69). При этом счетчик барьера декрементируется. Если счетчик больше `0`, этот поток `jthread` блокируется. Если счетчик равен `0`, выполняется функция завершения, которая перемещает следующий автомобиль на участок покраски или останавливает конвейер. Выполнив функцию завершения, барьер сбрасывает свой счетчик и разблокирует ожидающие потоки `jthread`, чтобы они могли продолжить работу.
- Чтобы начать моделирование, в строке 76 в функции `main` первый автомобиль устанавливается на участок покраски, а в строках 80–81 запускаются два `jthread`, имитирующие покрасочных роботов.

17.14. Семафоры (C++20)

Еще один механизм взаимного исключения потоков — это **семафор (semaphore)**, который Эдсгер Дейкстра описал в своей фундаментальной работе о взаимодействующих последовательных процессах[1, 2, 3]. Семафор содержит целое число, обозначающее максимальное количество конкурентных потоков, которые могут получать доступ к общему ресурсу, например к общим изменяемым данным. После инициализации это целое число может быть прочитано или изменено только двумя операциями: *P* и *V* (от голландских слов «proberen» — «проверять» и «verhogen» — «увеличивать»)[4]. Поток вызывает операцию *P* (также называемую операцией ожидания), когда хочет войти в критическую секцию, или вызывает операцию *V* (также называемую операцией сигнализации), когда хочет выйти из критической секции. Если в критической секции работает максимальное количество потоков, другие потоки, которые хотят войти в критическую секцию, должны ждать. Операции *P* и *V* — это абстракции, которые инкапсулируют детали реализации взаимного исключения. Они могут поддерживать любое количество взаимодействующих потоков. В классах семафоров C++ эти операции называются **захватом (acquire)** и **освобождением (release)**. Основные принципы семафоров хорошо изложены в книге Аллена Б. Дауни «The Little Book of Semaphores»[5].

Заголовок `<semaphore>` (C++20) содержит средства для реализации **семафоров со счетчиком (counting semaphores)** и **бинарных семафоров (binary semaphores)**:

- Класс `std::counting_semaphore` реализует концепцию семафора со счетчиком, более низкоуровневого, чем `std::latch` и `std::barrier`, но все же более высокоуровневого, чем мьютексы, блокировки, условные переменные и атомарные типы[6, 7]. Класс `counting_semaphore` позволяет нескольким потокам получать доступ к общему ресурсу. Его конструктор инициализирует внутренний счетчик семафора — целое число. Когда любой из потоков захватывает

---

[1] Э. В. Дейкстра (E. W. Dijkstra), «Cooperating Sequential Processes», Технологический университет, Эйндховен, Нидерланды, 1965 г., перепечатано в F. Genuys, ed., Programming Languages, с. 43–112. Нью-Йорк: Academic Press, 1968 г.

[2] Харви Дейтел (Harvey Deitel), Пол Дейтел (Paul Deitel) и Дэвид Чоффнес (David Choffnes), глава 5, «Asynchronous Concurrent Execution». «Operating Systems», 3-е изд., с. 227–233. Upper Saddle River, NJ: Prentice Hall, 2004 г.

[3] «Semaphore (Programming)». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Semaphore_(programming).

[4] «Semaphore (Programming) — Operation Names». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Semaphore_(programming)#Operation_names.

[5] Аллен Б. Дауни (Allen B. Downey), «The Little Book of Semaphores» (версия 2.2.1), раздел «3.6.4. Barrier Solution», 2016. Green Tea Press. https://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf. Лицензия: http://creativecommons.org/licenses/by-nc-sa/4.0. Спасибо одному из наших рецензентов, Энтони Уильямсу, за то, что указал нам на работу Дауни.

[6] Стандарт C++, «Thread Support Library — Semaphore». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/thread.sema.

[7] «std::counting_semaphore, std::binary_semaphore». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/thread/counting_semaphore.

семафор, счетчик декрементируется на 1. Пока счетчик больше 0, общий ресурс остается доступным. Если счетчик уменьшился до 0, все потоки, пытающиеся захватить семафор, блокируются. Когда поток освобождает семафор, счетчик инкрементируется на 1 (по умолчанию), а потоки, ожидающие захвата семафора, разблокируются.

- Класс `std::binary_semaphore` реализует концепцию бинарного семафора. Он представляет собой `counting_semaphore` с максимальным значением счетчика 1. Его можно использовать подобно классу `std::mutex`[1].

Синхронизация производителя и потребителя с помощью std::binary_semaphore (C++20)

В стандарте C++ говорится, что семафоры «широко используются для реализации других примитивов синхронизации и в тех случаях, когда они применимы, могут быть более эффективными, чем условные переменные»[2]. На илл. 17.16 показан класс `SynchronizedBuffer`, реализованный с использованием бинарных семафоров для взаимного исключения потоков. Это аккуратный, простой и высокоуровневый способ заменить низкоуровневый код с мьютексами. Мы используем функцию `main` из кода на илл. 17.8, поэтому не повторяем ее здесь.

```
// Fig. 17.16: SynchronizedBuffer.h
// Класс SynchronizedBuffer использует два семафора binary_semaphore для
// синхронизации доступа двух потоков к общей изменяемой переменной типа int
#pragma once
#include <fmt/format.h>
#include <iostream>
#include <semaphore>
#include <string>

using namespace std::string_literals;

class SynchronizedBuffer {
public:
   // Помещаем значение в m_buffer
   void put(int value) {
      // Захватываем семафор, чтобы получить право записи в m_buffer
      m_produce.acquire(); // Блокируем, если это не очередь производителя

      m_buffer = value; // Записываем в m_buffer
      m_occupied = true;

      std::cout << fmt::format("{:<40}{}\t\t{}\n",
         "Producer writes "s + std::to_string(value),
         m_buffer, m_occupied);

      m_consume.release(); // Разрешаем потребителю читать
   }
```

Окончание ⇨

[1] «std::counting_semaphore, std::binary_semaphore».

[2] Стандарт C++, «32.7. Semaphore». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/thread.sema.

```
28
29    // Возвращаем значение из m_buffer
30    int get() {
31       int value; // Для хранения значения, возвращенного функцией get
32
33       // Захватываем семафор, чтобы получить право чтения из m_buffer
34       m_consume.acquire(); // Блокируем, если это не очередь потребителя
35
36       value = m_buffer; // Читаем из m_buffer
37       m_occupied = false;
38
39       std::cout << fmt::format("{:<40}{}\t\t{}\n",
40          "Consumer reads "s + std::to_string(m_buffer),
41          m_buffer, m_occupied);
42
43       m_produce.release(); // Разрешаем производителю записывать
44       return value;
45    }
46 private:
47    std::binary_semaphore m_produce{1}; // Производитель может производить
48    std::binary_semaphore m_consume{0}; // Потребитель не может потреблять
49    bool m_occupied{false};
50    int m_buffer{-1}; // Общий буфер для потоков производителя и потребителя
51 };
```

```
Operation                       Buffer      Occupied
---------                       ------      --------
Producer writes 1               1           true
Consumer reads 1                1           false
Producer writes 2               2           true
Consumer reads 2                2           false
Producer writes 3               3           true
Consumer reads 3                3           false
Producer writes 4               4           true
Consumer reads 4                4           false
Producer writes 5               5           true
Consumer reads 5                5           false
Producer writes 6               6           true
Consumer reads 6                6           false
Producer writes 7               7           true
Consumer reads 7                7           false
Producer writes 8               8           true
Consumer reads 8                8           false
Producer writes 9               9           true
Consumer reads 9                9           false
Producer writes 10              10          true
Producer done producing
Terminating Producer
Consumer reads 10               10          false
Consumer read values totaling 55
Terminating Consumer
```

Илл. 17.16 | Класс SynchronizedBuffer использует два семафора binary_semaphore для синхронизации доступа двух потоков к общей изменяемой переменной типа int

В коде, показанном на илл. 17.16, используются два семафора `std::binary_semaphore` для координации потоков производителя и потребителя:

- семафор `m_produce` (строка 47) инициализируется значением `1`, указывающим на то, что производитель изначально может выполнить запись, поскольку `m_buffer` пуст;
- семафор `m_consume` (строка 48) инициализируется значением `0`, указывающим на то, что потребитель изначально не может выполнить чтение, поскольку `m_buffer` пуст.

В этом примере переменная `m_occupied` (строка 49) нужна только для вывода данных и не играет никакой роли в синхронизации потоков.

Функция put класса SynchronizedBuffer

Логика класса `SynchronizedBuffer` стала проще благодаря семафорам `std::binary_semaphore`. Когда поток-производитель вызывает функцию `put` (определена в строках 15–27), в строке 17 вызывается функция `acquire` семафора `m_produce`. Если счетчик семафора `m_produce` равен `0`, поток-производитель будет заблокирован, пока счетчик не увеличится до `1`. Если счетчик равен `1`, поток-производитель захватывает семафор, уменьшает его счетчик до `0` и записывает произведенное значение в `m_buffer`. Производитель не может производить снова, пока счетчик `m_produce` не станет равен `1`, что произойдет только тогда, когда потребитель прочитает значение из буфера.

Когда производитель завершает обновление буфера, в строке 26 вызывается функция `release` семафора `m_consume`, что увеличивает счетчик семафора до `1`. Если поток-потребитель заблокирован в ожидании захвата `m_consume`, он разблокируется, чтобы продолжить работу. В противном случае, когда потребитель пытается вызвать `get`, он может немедленно захватить `m_consume` и продолжить работу.

Функция get класса SynchronizedBuffer

Когда поток-потребитель вызывает функцию `get` (определенную в строках 30–45), в строке 34 вызывается функция `acquire` семафора `m_consume`. Если счетчик семафора `m_consume` равен `0`, поток-потребитель блокируется до тех пор, пока счетчик не станет равен `1`. Если счетчик равен `1`, поток-потребитель захватывает `m_consume`, уменьшает его счетчик до `0`, затем читает значение из `m_buffer`. Потребитель не может снова потреблять, пока счетчик `m_consume` не станет равен `1`, что произойдет только тогда, когда производитель запишет следующее значение в буфер.

Когда потребитель завершает чтение из буфера, в строке 43 вызывается функция `release` семафора `m_produce`, что увеличивает счетчик семафора до `1`. Если поток производителя заблокирован в ожидании захвата `m_produce`, он разблокируется, чтобы продолжить работу. В противном случае, когда производитель пытается вызвать `put`, он может немедленно захватить `m_produce` и продолжить работу.

17.15. C++23: взгляд в будущее конкурентности

Многие новые средства конкурентности рассматриваются для включения в будущие версии C++. Здесь мы приводим обзор некоторых из них и даем ссылки на источники дополнительной информации[1, 2].

17.15.1. Параллельные алгоритмы std::ranges

Комитет по стандартам C++ работает над параллельными версиями алгоритмов `std::ranges`, введенных в C++20. В документе «A Plan for C++23 Ranges» отмечаются существующие проблемы реализации, связанные с исполнителями[3].

17.15.2. Конкурентные контейнеры

На протяжении всей книги вы встречали различные контейнеры, не имеющие средств конкурентного доступа. В этой главе вы изучали класс `CircularBuffer`, в котором мы реализовали конкурентный доступ с помощью `std::mutex`, `std::unique_lock` и `std::condition_variable`. Но вообще мы рекомендуем вместо создания собственных контейнеров с конкурентным доступом использовать уже существующие, которые управляют синхронизацией без вашего участия. К ним относятся, в частности, **конкурентные очереди (concurrent queues)** и **конкурентные карты (concurrent maps)**, также называемые **конкурентными хеш-таблицами (concurrent hash tables)**, или **конкурентными словарями (concurrent dictionaries)**. Они написаны экспертами, тщательно протестированы и отлажены, эффективно работают и помогут вам избежать распространенных «ловушек» и «подводных камней». Возможно, такие контейнеры будут включены в стандартную библиотеку C++23. Пока же вам придется использовать контейнеры из сторонних библиотек, таких как Google Concurrency Library (GCL)[4] или Microsoft Parallel Patterns Library[5]. Дополнительные сведения о параллельных очередях опубликованы в документе «C++ Concurrent Queues»[6]. Там сообщается, что реализация параллельных очередей обеспечивается классами `buffer_queue` и `lock_free_buffer_queue` из GCL. Дополнительные сведения о предварительных реализациях параллельных карт опубликованы в документах Комитета по стандартам C++:

---

1 «C++23». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/C++23.

2 Вилле Вутилайнен (Ville Voutilainen), «To Boldly Suggest an Overall Plan for C++23», 25 ноября 2019 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p0592.

3 Барри Ревзин (Barry Revzin), Конор Хукстра (Conor Hoekstra) и Тим Сонг (Tim Song), «A Plan for C++23 Ranges», 14 октября 2020 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p2214.

4 Аласдер Макинтош (Alasdair Mackintosh), «Google Concurrency Library (GCL)». Был доступен: 15 сентября 2023 г. https://github.com/alasdairmackintosh/google-concurrency-library.

5 «Parallel Patterns Library (PPL) — Parallel Containers and Objects». Был доступен: 15 сентября 2023 г. https://docs.microsoft.com/en-us/cpp/parallel/concrt/parallel-containers-and-objects.

6 Лоуренс Кроул (Lawrence Crowl) и Крис Майсен (Chris Mysen), «C++ Concurrent Queues». Был доступен: 15 сентября 2023 г. https://wg21.link/p0260.

- «Concurrent Associative Data Structure with Unsynchronized View»[1];
- «Concurrent Map Customization Options (SG1 Version)»[2].

Здесь вы найдете предварительную реализацию конкурентной карты:

https://github.com/BlazingPhoenix/concurrent-hash-map

17.15.3. Другие материалы, связанные с конкурентностью

В следующих документах Комитета по стандартам C++ рассматриваются предложения по введению дополнительных средств конкурентности в C++23 и последующие версии:

- В документах «Hazard Pointers»[3] и «Concurrent Data Structures: Read-Copy-Update»[4] предложены средства безопасного возврата ресурсов, разделяемых между потоками.
- В документе «apply() for synchronized_value<T>»[5] описано усовершенствование предыдущего предложения, основанное на классе `synchronized_value<T>`, который автоматически использует мьютекс для синхронизации конкурентного доступа к объектам типа `T`. Предлагаемая функция `apply` будет принимать в качестве аргументов функцию и один или несколько объектов класса `synchronized_value<T>`. Затем она будет вызывать полученную в аргументе функцию для каждого объекта `synchronized_value<T>`, используя соответствующий мьютекс для синхронизации доступа конкурентных потоков.

17.16. Итоги

В этой главе были представлены современные стандартные средства конкурентного программирования на C++, предназначенные для повышения быстродействия

---

[1] Сергей Мурылев, Антон Малахов и Энтони Полухин, «Concurrent Associative Data Structure with Unsynchronized View», 13 июня 2019 г. Был доступен: 15 сентября 2023 г. http://wg21.link/p0652.

[2] Дэвид Голдблатт (David Goldblatt), «Concurrent Map Customization Options (Sg1 Version)», 16 июня 2019 г. Был доступен: 15 сентября 2023 г. https://wg21.link/P1761.

[3] Магед М. Микаэль (Maged M. Michael), Майкл Вонг (Michael Wong), Пол Маккенни (Paul McKenney), Джеффри Ромер (Geoffrey Romer), Эндрю Хантер (Andrew Hunter), Артур О'Двайр (Arthur O'Dwyer), Дейзи С. Холлман (Daisy S. Hollman), Ж. Ф. Бастьен (J. F. Bastien), Ханс Бом (Hans Boehm), Дэвид Голдблатт (David Goldblatt), Франк Бирбахер (Frank Birbacher), Матиас Стерн (Mathias Stearn) и Йенс Маурер (Jens Maurer), «Hazard Pointers», 9 апреля 2021 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p1121.

[4] Пол Маккенни, Майкл Вонг, Магед М. Микаэль, Джеффри Ромер, Эндрю Хантер, Артур О'Двайр, Дейзи С. Холлман, Ж. Ф. Бастьен, Ханс Бом, Дэвид Голдблатт, Франк Бирбахер, Эрик Ригторп (Erik Rigtorp), Томаш Камински (Tomasz Kaminski) и Йенс Маурер, «Concurrent Data Structures: Read-Copy-Update», 14 мая 2021 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p1122.

[5] Энтони Уильямс (Anthony Williams), «apply() for synchronized_value<T>», 2 марта 2017 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p0290.

программ в многоядерных системах. Мы сравнили последовательное, конкурентное и параллельное выполнение программы и показали пример диаграммы жизненного цикла потока. Мы объяснили, почему писать конкурентный код очень сложно и почему при этом часто возникают ошибки. Как правило, разработчикам следует использовать простые и удобные высокоуровневые средства конкурентности из стандартной библиотеки, на которых мы акцентировали внимание в этой главе.

Мы показали, что высокоуровневые параллельные алгоритмы (C++17) автоматически распределяют задачи между несколькими потоками для повышения быстродействия. С помощью средств измерения интервалов времени, имеющихся в заголовке `<chrono>`, мы сравнили скорость работы последовательных и параллельных алгоритмов в многоядерных системах. Мы убедились, что параллельная версия алгоритма `std::sort`, вызванная с политикой выполнения `std::execution::par`, сортирует большие наборы данных значительно быстрее, чем последовательная версия этого алгоритма. Мы также сравнили быстродействие алгоритма `std::tranform`, вызванного с политиками выполнения `std::execution::par` (для параллельных операций) и `std::execution::unseq` (для векторных операций).

Мы выполняли задачи в отдельных потоках с помощью класса `std::jthread` (C++20), который в C++ Core Guidelines рекомендуется вместо класса `std::thread` (C++11). Мы использовали встроенную в `jthread` поддержку кооперативного прерывания, чтобы корректно прерывать потоки, давая им возможность завершать критическую часть работы и правильно освобождать ресурсы.

Мы представили отношения между производителем и потребителем, актуальные для конкурентного программирования, и продемонстрировали проблемы, связанные с одновременным доступом потока-производителя и потока-потребителя к общим изменяемым данным без синхронизации. Мы решили эти проблемы, синхронизировав доступ к общим изменяемым данным с помощью низкоуровневых примитивов конкурентности `std::mutex`, `std::condition_variable` и `std::unique_lock` (C++11). Затем мы использовали эти примитивы для реализации синхронизированного кольцевого буфера, чтобы минимизировать периоды ожидания потоков и повысить быстродействие.

Мы использовали шаблон функции `std::async` и шаблон класса `std::future` (C++11) для неявного создания потоков, асинхронного выполнения задач и передачи результатов каждой задачи обратно в поток, вызвавший функцию `async`. Мы обсудили, как «под капотом» функция `async` использует объект `std::promise` для межпоточной коммуникации, чтобы вернуть результат задачи или исключение потоку, вызвавшему `async`.

Мы представили вам атомарные типы (C++11), которые позволяют потокам очень просто обрабатывать изменяемые данные без явной синхронизации и блокировки. Мы рассмотрели несколько улучшений атомарных типов в C++20 и отметили, что в стандартной библиотеке C++20 для Visual C++, g++ и clang++ атомарные типы используются для реализации высокоуровневых примитивов `std::latch` и `std::barrier`, а также семафоров.

Мы продемонстрировали примитивы координации потоков `std::latch` и `std::barrier`, показав, что для их корректной работы не нужны мьютексы и блокировки. Далее мы использовали семафоры C++20 без мьютексов, блокировок и условных переменных, чтобы синхронизировать доступ к общим изменяемым данным.

Наконец, мы упомянули несколько высокоуровневых средств конкурентности, в том числе конкурентные контейнеры и параллельные версии алгоритмов `std::ranges`, которые могут войти в состав C++23 или более поздних версий. Большинство программистов предпочитают использовать готовые конкурентные контейнеры, такие как конкурентные очереди и карты, в которых инкапсулирована синхронизация. Они помогают избежать распространенных «ловушек» и ошибок, возникающих при работе с низкоуровневыми примитивами синхронизации. Такие конкурентные контейнеры пока не входят в стандартную библиотеку C++, поэтому вам нужны сторонние библиотеки, такие как Google Concurrency Library и Microsoft Parallel Patterns Library.

В следующей главе мы подробно рассмотрим еще одно средство конкурентности: корутины. Это последнее из «большой четверки» новшеств C++20 (наряду с диапазонами, концептами и модулями). Вы будете использовать высокоуровневый подход, основанный на готовых библиотеках, для удобного создания корутин, которые позволят вам создавать сложные программы с конкурентными задачами, сохраняя простой последовательный стиль кодирования.

18

Корутины (C++20)

В этой главе вы:

- узнаете, что такое корутины и как с ними работать;
- используете ключевое слово `co_yield`, чтобы приостанавливать работу корутины-генератора и возвращать ее результат;
- используете оператор `co_await`, чтобы приостанавливать работу корутины, которая ожидает результата;
- используете инструкцию `co_return`, чтобы завершать работу корутины и возвращать ее результат (или просто управление) в точку вызова;
- используете библиотеку `generator` с открытым исходным кодом для простого создания корутины-генератора, содержащей `co_yield`;
- используете библиотеку `concurrencpp` с открытым исходным кодом для простого создания корутин, содержащих `co_await` и `co_return`;
- узнаете о тех возможностях корутин, которые планируются для будущих версий C++.

18.1. Введение

Если в программе есть долго выполняющаяся задача, то функция, вызываемая синхронно (выполняющая задачи по очереди), обычно запускает эту долго выполняющуюся задачу **асинхронно (asynchronously)**. Как правило, программа предоставляет этой асинхронной задаче **функцию обратного вызова (callback function)**, которая будет вызвана после завершения задачи (это может быть не только функция, но и лямбда или функциональный объект). Такой стиль кодирования упрощается благодаря **корутинам (coroutines)**, которые также называют **сопроцедурами**, или **сопрограммами**. Корутины — последнее из «большой четверки» новшеств C++20 (диапазоны, концепты, модули и корутины), которые мы рассматриваем в этой книге.

Корутина[1] — это функция, которая может приостанавливать свое выполнение и продолжать его позже. Компилятор полностью создает вместо вас код, реализующий следующие механизмы:

- приостановка корутины и возврат управления в точку вызова;
- последующее продолжение работы приостановленной корутины.

Вы увидите, что функция, содержащая любое из ключевых слов `co_await`, `co_yield` или `co_return`, является корутиной.

Корутины дают вам возможность программировать конкурентные задачи, сохраняя простой последовательный стиль кодирования. Эта возможность основана на сложной инфраструктуре. Вы могли бы написать такую инфраструктуру своими силами, но это непросто, утомительно и чревато ошибками. Большинство экспертов рекомендует использовать готовые высокоуровневые библиотеки поддержки корутин. Мы продемонстрируем именно такой подход. Сообщество разработчиков с открытым исходным кодом предлагает несколько экспериментальных библиотек, предназначенных для быстрого и удобного программирования с корутинами. В разделе 18.2 перечислены некоторые из этих библиотек, в том числе две, используемые в наших примерах.

Когда нужны корутины

Перечислим некоторые сферы применения корутин[2]:

- обработка запросов веб-серверами;
- пошаговое выполнение функций в видеоиграх (по одному шагу на каждый кадр анимации);

---

[1] Термин «coroutine» ввел Мелвин Конвей (Melvin Conway) в 1958 г. Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Coroutine.

[2] Джеффри Ромер (Geoffrey Romer), Гор Нишанов (Gor Nishanov), Льюис Бейкер (Lewis Baker) и Михаил Михайлов, «Coroutines: Use-Cases and Trade-Offs», 19 февраля 2019 г. Был доступен: 15 сентября 2023 г. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1493r0.pdf.

- получение данных (например, результатов длительных вычислений; раздел 18.5) после того, как они становятся доступными, в том числе при обработке данных, генерируемых устройствами «интернета вещей» (IoT) и при загрузке больших файлов[1];
- функции, лениво вычисляющие последовательности значений (такие функции называют генераторами, см. раздел 18.4) и предоставляющие каждое очередное значение по запросу[2];
- событийно-ориентированное кодирование без функций обратного вызова[3], например, в симуляциях, пользовательских интерфейсах, серверах, играх, неблокирующих операциях ввода-вывода[4];
- кооперативная многозадачность[5, 6];
- структурированная конкурентность[7];
- программирование реактивных потоков[8];
- реализация машин состояний[9].

18.2. Библиотеки поддержки корутин

Для поддержки корутин необходимы специализированные классы с многочисленными функциями и вложенными типами данных. Вы потратили бы много времени и, вероятнее всего, допустили бы ошибки, пробуя написать такие сложные классы самостоятельно. В стандартной библиотеке C++20 есть только

---

1 «What Are Use Cases for Coroutines?». Был доступен: 15 сентября 2023 г. https://stackoverflow.com/questions/303760/what-are-use-cases-for-coroutines.

2 «Coroutines». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/language/coroutines.

3 Давид Мазьер (David Mazières), «My Tutorial and Take on C++20 Coroutines», февраль 2021 г. Был доступен: 15 сентября 2023 г. https://www.scs.stanford.edu/~dm/blog/c++-coroutines.html.

4 Techmunching, «Coroutines and Their Introduction in C++», 30 мая 2020 г. Был доступен: 15 сентября 2023 г.

5 Райнер Гримм (Rainer Grimm), «C++20: More Details to Coroutines», 27 марта 2020 г. Был доступен: 15 сентября 2023 г. http://modernescpp.com/index.php/component/content/article/54-blog/c-20/488-c-20-coroutines-more-details.

6 Бобби Приамбодо (Bobby Priambodo), «Cooperative vs. Preemptive: a Quest to Maximize Concurrency Power», 3 сентября 2019 г. Был доступен: 15 сентября 2023 г. https://medium.com/traveloka-engineering/cooperative-vs-preemptive-a-quest-to-maximize-concurrency-power-3b10c5a920fe.

7 Льюис Бейкер (Lewis Baker), «Structured Concurrency: Writing Safer Concurrent Code with Coroutines and Algorithms», 14 октября 2019 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=1Wy5sq3s2rg.

8 Джефф Томас (Jeff Thomas), «Exploring Coroutines», April 7, 2021. Был доступен: 15 сентября 2023 г. https://blog.coffeetocode.com/2021/04/exploring-coroutines.

9 Стив Дауни (Steve Downey), «Converting a State Machine to a C++ 20 Coroutine», 29 июня 2021 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=Z8jHi9Cs6Ug.

низкоуровневые примитивы, необходимые разработчикам таких классов, то есть библиотек поддержки корутин. Льюис Бейкер, который работал над несколькими такими библиотеками, сказал, что эти примитивы «можно рассматривать как низкоуровневый язык ассемблера для корутин»[1]. Как правило, программисты предпочитают использовать готовые библиотеки поддержки корутин. Поддержка корутин в стандартной библиотеке C++ ожидается в C++23[2,3].

За последние годы в сообществе разработчиков с открытым исходным кодом созданы несколько нестандартных экспериментальных библиотек поддержки корутин в C++. Некоторые из них заслуживают особого внимания:

- Библиотека Льюиса Бейкера `cppcoro`[4] упоминается во многих книгах, видеороликах и сообщениях в блогах, но больше не поддерживается[5].
- Библиотека `folly::coro` является подмножеством популярной большой библиотеки утилит `folly` с открытым исходным кодом[6]. Льюис Бейкер входит в команду разработчиков `folly::coro`. В документации к этой библиотеке указано, что она является экспериментальной. К сожалению, `folly::coro` не может быть установлена независимо от `folly`.
- Библиотека `concurrencpp`[7] Дэвида Хайма активно поддерживается. Она позволяет удобно разрабатывать корутины с ключевыми словами `co_await` и `co_return`. На момент написания этой книги в библиотеке `concurrencpp` еще не было поддержки `co_yield`, но Дэвид добавил ее в версии 0.1.4[8].
- Библиотека `generator`[9] Сая Бренда — это заголовочная библиотека для разработки корутин-генераторов, возвращающих каждое очередное значение по инструкции `co_yield`. Генератор производит значения по требованию, это называется ленивыми вычислениями. Они отличаются от жадных вычислений, примером которых является алгоритм `std::ranges::generate` (раздел 14.4.1), сразу генерирующий значения и записывающий их в элементы диапазона, на-

---

[1] Льюис Бейкер (Lewis Baker), «C++ Coroutines: Understanding Operator co_await», 17 ноября 2017 г. Был доступен: 15 сентября 2023 г. https://lewissbaker.github.io/2017/11/17/understanding-operatorco-await.

[2] Вилле Вутилайнен (Ville Voutilainen), «To Boldly Suggest an Overall Plan for C++23», 25 ноября 2019 г. Был доступен: 15 сентября 2023 г. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0592r4.html.

[3] «C++23». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/C++23.

[4] Льюис Бейкер (Lewis Baker), «cppcoro». Был доступен: 15 сентября 2023 г. https://github.com/lewissbaker/cppcoro.

[5] В письме от 18 сентября 2021 г. Льюис Бейкер сообщил, что библиотека cppcoro больше не поддерживается. Сейчас он работает над экспериментальными библиотеками folly::coro и libunifex.

[6] «folly». Был доступен: 15 сентября 2023 г. https://github.com/facebook/folly.

[7] Дэвид Хайм (David Haim), «concurrencpp». Был доступен: 15 сентября 2023 г. https://github.com/David-Haim/concurrencpp.

[8] Мы общались с Давидом Хаймом по электронной почте 14 ноября 2021 г.

[9] Сай Бренд (Sy Brand), «generator». Был доступен: 15 сентября 2023 г. https://github.com/TartanLlama/generator.

пример `vector`. Для создания диапазона с большим количеством элементов требуется много памяти и времени. Если программе не нужны все значения сразу, генератор предпочтительнее, так как он может уменьшить потребление памяти и повысить быстродействие. Кроме того, генераторы могут создавать бесконечные последовательности, такие как числа Фибоначчи (раздел 18.4) или простые числа[1].

Мы познакомим вас с корутинами на примере библиотек `concurrencpp` и `generator`[2]. Библиотека `concurrencpp` легко устанавливается, активно поддерживается и имеет понятную документацию с множеством примеров кода. В нее входят высокоуровневые средства конкурентности, в том числе:

- задачи для асинхронного выполнения функций;
- **исполнители (executors)** для планирования задач (возможно, выполняющие их в отдельных потоках);
- таймеры для выполнения задач в будущем;
- разные полезные функции.

В примере кода из раздела 18.4 мы используем библиотеку `generator` для реализации корутины-генератора, которая генерирует по запросу каждое очередное число Фибоначчи и возвращает его вызывающей стороне через `co_yield`. В примере кода из раздела 18.5 представлены задачи и исполнители библиотеки `concurrencpp` (их стандартизированные версии ожидаются в C++23). В примере кода из раздела 18.6 они используются для создания корутины, демонстрирующей `co_await` и `co_return`.

18.3. Установка библиотек concurrencpp и generator

Библиотека concurrencpp

Чтобы установить `concurrencpp`, следуйте инструкциям для вашей платформы, опубликованным по этому адресу:

https://github.com/David-Haim/concurrencpp#building-installing-and-testing

Если вы работаете с Visual C++, выполните следующие дополнительные действия:

1. Откройте файл `concurrencpp.sln` из библиотеки `concurrencpp\build\lib` и соберите решение (build solution) для генерации файлов библиотеки.

---

[1] Visual C++ предоставляет реализацию std::experimental::generator в заголовке <experimental/generator>.

[2] Спасибо Энтони Уильямсу (Anthony Williams, https://www.linkedin.com/in/anthonyajwilliams), автору книги «C++ Concurrency in Action», 2-е изд. (https://www.manning.com/books/c-plus-plusconcurrency-in-action-second-edition), за то, что поделился с нами своими мыслями, когда мы решали, что включить в этот раздел о корутинах.

2. В вашем решении Visual Studio, которое будет использовать `concurrencpp`, следуйте инструкциям из раздела 3.12, чтобы добавить папку `concurrencpp\include` в путь к включаемым заголовкам.
3. Далее выберите `File > Add > Existing Project...`, перейдите в папку библиотеки `concurrencpp\build\lib` и добавьте файл `concurrencpp.vcxproj` к вашему решению.
4. Наконец, щелкните правой кнопкой мыши на своем проекте, выберите `Add > Reference...`, затем в диалоговом окне `Add Reference` отметьте проект `concurrencpp` и нажмите `OK`.

Библиотека generator

Загрузите заголовочную библиотеку `generator` по этому адресу:

https://github.com/TartanLlama/generator

Затем включите ее в свой проект. Если хотите, вы можете клонировать репозиторий GitHub, а затем добавить папку `include` библиотеки в путь к включаемым заголовкам вашего компилятора.

18.4. Создание корутины-генератора с помощью библиотеки generator

Генератор — это корутина, производящая значения по требованию. Когда вы вызываете корутину-генератор, она использует выражение `co_yield`, чтобы приостановить свое выполнение и вернуть очередное сгенерированное значение в точку вызова. Ожидается, что поддержка генераторов войдет в стандартную библиотеку C++23. Для этого примера мы будем использовать библиотеку `generator`[1]. Ее шаблон класса `tl::generator` позволяет указать возвращаемый тип корутины-генератора. Библиотека предоставляет механизмы, позволяющие выражению `co_yield` возвращать значения в точку вызова корутины-генератора.

В показанной на илл. 18.1 программе определена корутина-генератор, которая производит ряд чисел Фибоначчи:

`0, 1, 1, 2, 3, 5, 8, 13, 21, ...`

Напомним, что этот ряд начинается с чисел `0` и `1`, а каждое последующее число является суммой двух предыдущих.

```
1 // fig18_01.cpp
2 // Создание корутины-генератора, содержащей co_yield
3 #include <fmt/format.h>
4 #include <iostream>
5 #include <sstream>
```

Окончание ⇨

[1] На момент написания этой книги библиотека generator не работает с clang++.

```
#include <thread>
#include <tl/generator.hpp>

// Получаем идентификатор текущего потока в виде строки
std::string id() {
   std::ostringstream out;
   out << std::this_thread::get_id();
   return out.str();
}

// Корутина, многократно возвращающая следующее число из ряда Фибоначчи
tl::generator<int> fibonacciGenerator(int limit) {
   std::cout << fmt::format(
      "Thread {}: fibonacciGenerator started executing\n", id());

   int value1{0}; // Fibonacci(0)
   int value2{1}; // Fibonacci(1)

   for (int i{0}; i < limit; ++i) {
      co_yield value1; // Возвращаем текущее значение переменной value1

      // Обновляем value1 и value2 для следующей итерации
      int temp{value1 + value2};
      value1 = value2;
      value2 = temp;
   }

   std::cout << fmt::format(
      "Thread {}: fibonacciGenerator finished executing\n", id());
}

int main() {
   std::cout << fmt::format("Thread {}: main begins\n", id());

   // Выводим на экран первые 10 чисел Фибоначчи
   for (int i{0}; auto value : fibonacciGenerator(10)) {
      std::cout << fmt::format("Fibonacci({}) is {}\n", i++, value);
   }

   std::cout << fmt::format("Thread {}: main ends\n", id());
}
```

```
Thread 7316: main begins
Thread 7316: fibonacciGenerator started executing
Fibonacci(0) is 0
Fibonacci(1) is 1
Fibonacci(2) is 1
Fibonacci(3) is 2
Fibonacci(4) is 3
Fibonacci(5) is 5
Fibonacci(6) is 8
```

```
Fibonacci(7) is 13
Fibonacci(8) is 21
Fibonacci(9) is 34
Thread 7316: fibonacciGenerator finished executing
Thread 7316: main ends
```

Илл. 18.1 | Создание корутины-генератора, содержащей co_yield

Функция id, преобразующая идентификатор потока в std::string

Функция `id` (строки 10–14) использует перегруженный оператор `<<` объекта `std::thread::id` для преобразования идентификатора потока в строку `std::string`. Этот идентификатор нужен, чтобы показать, что `main` и `fibonacciGenerator` выполняются в одном и том же потоке.

Корутина fibonacciGenerator

В строках 17–35 определена корутина `fibonacciGenerator`. Параметром `limit` этой функции задается количество чисел Фибоначчи, которые надо вычислить, начиная с `0`. В строках 18–19 выводится на экран идентификатор потока, в котором выполняется `fibonacciGenerator`. Выходные данные программы показывают, что идентификатор этого потока совпадает с идентификатором потока `main`, то есть корутина выполняется в том же потоке, что и `main`. В строках 21 и 22 определены переменные `value1` и `value2`. Изначально в них хранятся первые два числа ряда Фибоначчи. Когда программа запрашивает у генератора новое значение, инструкция `co_yield` (строка 25) приостанавливает выполнение корутины и немедленно возвращает очередное число Фибоначчи. Когда программа впоследствии запрашивает следующее значение, выполнение корутины возобновляется и в строках 28–30 обновляются значения `value1` и `value2`. Затем в следующей итерации цикла инструкция `co_yield` возвращает следующее число Фибоначчи, снова приостанавливая выполнение корутины и возвращая это число в точку вызова. Так продолжается до тех пор, пока не будет получено последнее из требуемых чисел Фибоначчи. Затем в строках 33–34 снова выводится на экран идентификатор потока, в котором выполняется `fibonacciGenerator`. Мы видим, что это по-прежнему тот же поток, в котором выполняется `main`.

Функция main

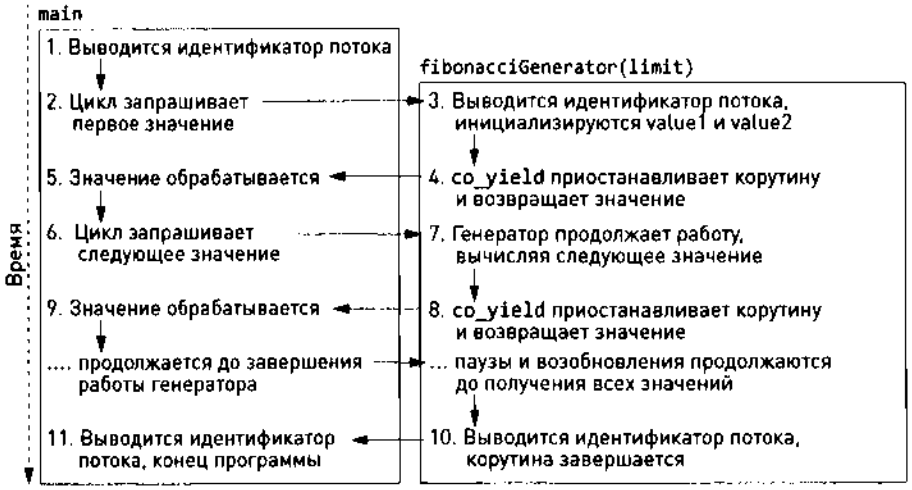
В строке 38 выводится на экран идентификатор потока `main`, поэтому мы можем подтвердить, что `main` и `fibonacciGenerator` выполняются в одном потоке. В строках 41–43 запрашиваются первые `10` чисел ряда Фибоначчи. Вызов `fibonacciGenerator(10)` возвращает `tl::generator<int>`, который предоставляет итераторы, поэтому его можно использовать в цикле `for` для диапазона значений. Когда каждая итерация цикла запрашивает очередное число, корутина `fibonacciGenerator` возобновляет выполнение для его вычисления, затем приостанавливается и возвращает результат. Наконец, в строке 45 выводится идентификатор потока `main` и сообщение о завершении `main`. Идентификаторы

потоков в выходных данных подтверждают, что `fibonacciGenerator` выполняет свою работу в потоке `main`.

Вызов корутины не приводит к автоматическому созданию нового потока. Если нужны другие потоки, корутина должна их создать, как мы сделаем в разделах 18.5 и 18.6. Вы можете создавать потоки и управлять ими, применяя методы, показанные в главе 17, или позволить таким библиотекам, как `concurrencpp`, создавать потоки и управлять ими без вашего участия.

Диаграмма потока управления программы с корутиной-генератором

Основной поток управления (flow of control) этой программы показан на диаграмме. Номера на диаграмме соответствуют номерам в ее описании:



1. Функция `main` выводит на экран идентификатор своего потока.
2. Цикл в функции `main` запрашивает у корутины `fibonacciGenerator` первое значение.
3. Корутина `fibonacciGenerator` выводит на экран идентификатор своего потока и инициализирует свои локальные переменные.
4. Корутина `fibonacciGenerator` получает значение, что приостанавливает ее выполнение и возвращает это значение и управление в функцию `main`.
5. Цикл в функции `main` обрабатывает полученное значение.
6. Цикл в функции `main` запрашивает у корутины `fibonacciGenerator` следующее значение.
7. Корутина `fibonacciGenerator` возобновляет выполнение и вычисляет следующее значение.

8. Корутина `fibonacciGenerator` возвращает значение. Это снова приостанавливает ее выполнение и возвращает значение и управление в функцию `main`.
9. Цикл в функции `main` обрабатывает полученное значение.
10. Шаги 6–9 продолжаются до тех пор, пока `fibonacciGenerator` не закончит генерацию значений. Затем `fibonacciGenerator` выводит на экран идентификатор своего потока, завершает работу и возвращает управление функции `main`.
11. Цикл в функции `main` завершается, затем функция `main` выводит на экран идентификатор своего потока и завершается.

Корутины в C++ являются бесстековыми

Компилятор управляет механизмами, позволяющими приостанавливать и возобновлять выполнение корутин. Он создает **состояние корутины (coroutine state)**[1], которое содержит информацию, необходимую для возобновления работы корутины. Это состояние хранится в куче (в динамически выделенной области памяти), а не в стеке, поэтому такие корутины называются бесстековыми[2]. Если компилятор выясняет, что время жизни корутины не превышает времени жизни вызывающей ее программы, он может избежать ресурсоемкого выделения памяти в куче[3]. Марк Грегуар (Marc Gregoire) отмечает, что «использование памяти для бесстековых корутин минимально, что позволяет миллионам или даже миллиардам корутин работать одновременно»[4].

18.5. Запуск задач с помощью библиотеки concurrencpp

Прежде чем мы приступим к реализации корутины с помощью `concurrencpp`, давайте используем эту библиотеку для планирования задач, выполняемых в отдельных потоках. Как и в случае с потоками стандартной библиотеки, каждая задача выполняет функцию, или лямбду, или функциональный объект. В показанной на илл. 18.2 программе используются четыре компонента `concurrencpp`:

- Класс `concurrencpp::runtime` управляет вашим взаимодействием с `concurrencpp`. Он предоставляет доступ к различным исполнителям для планирования задач. Он также обеспечивает правильное завершение исполнителями запланированных задач при уничтожении объектов `runtime`. Вы должны создать локальный объект `runtime`, обычно в начале функции `main` (строка 35).

---

[1] Стандарт C++, «Coroutine Definitions». Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861/dcl.fct.def.coroutine#9.

[2] Варун Рамеш (Varun Ramesh), «Stackless vs. Stackful Coroutines», 18 августа 2017 г. Был доступен: 15 сентября 2023 г. https://blog.varunramesh.net/posts/stackless-vs-stackful-coroutines.

[3] «Coroutines — Heap Allocation». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/language/coroutines#Heap_allocation.

[4] Марк Грегуар (Marc Gregoire), «Professional C++», 5-е изд., с. 963. Indianapolis, IN: John Wiley & Sons, 2021.

Когда `main` завершается, локальный объект `runtime` выходит из области видимости. Его деструктор останавливает исполнители и все оставшиеся запланированные задачи, еще не завершившиеся.

- Класс `concurrencpp::thread_pool_executor` (один из нескольких типов исполнителей) планирует выполнение задач. Он создает и координирует группу потоков, называемую **пулом потоков (thread pool)**[1], и назначает этим потокам задачи для выполнения. Этот исполнитель может повторно использовать существующие потоки пула, чтобы избежать ресурсоемкого создания новых потоков для новых задач. Он также может оптимизировать количество потоков, чтобы не допускать простоев процессора, но и не создавать чрезмерное количество потоков, из-за которых приложение исчерпает ресурсы системы. Стандартные исполнители ожидаются в C++23, но такие библиотеки, как `concurrencpp`, `libunifex`[2] и `folly::coro`, предлагают их уже сейчас.
- Класс `concurrencpp::task` представляет задачу для выполнения. Исполнители `concurrencpp` создают объекты этого класса, когда вы планируете задачи.
- Класс `concurrencpp::result` позволяет вам получить доступ к результату конкурентно выполняемой задачи или просто дождаться ее завершения, если она не возвращает никакого значения. Когда вы планируете `concurrencpp::task`, вы получаете `concurrencpp::result`.

```
// fig18_02.cpp
// Планирование задач с помощью concurrencpp::runtime
#include <chrono>
#include <concurrencpp/concurrencpp.h>
#include <fmt/format.h>
#include <iostream>
#include <random>
#include <sstream>
#include <thread>
#include <vector>

// Получаем идентификатор текущего потока в виде строки
std::string id() {
   std::ostringstream out;
   out << std::this_thread::get_id();
   return out.str();
}

// Функция printTask засыпает на заданное время (в миллисекундах);
// когда она возобновляет выполнение, она выводит свое имя и завершается
void printTask(std::string name, std::chrono::milliseconds sleep) {
   std::cout << fmt::format(
```

[1] «Thread pool». Был доступен: 15 сентября 2023 г. https://en.wikipedia.org/wiki/Thread_pool.

[2] «libunifex». Был доступен: 15 сентября 2023 г. https://github.com/facebookexperimental/libunifex.

```
23        "{} (thread ID: {}) going to sleep for {} ms\n",
24        name, id(), sleep.count());
25
26     // Переводим вызывающий поток в сон на sleep миллисекунд
27     std::this_thread::sleep_for(sleep);
28
29     std::cout << fmt::format("{} (thread ID: {}) done sleeping\n",
30        name, id());
31 }
32
33 int main() {
34     // Создаем объект runtime для планирования задач
35     concurrencpp::runtime runtime;
36
37     std::cout << fmt::format("main's thread ID: {}\n\n", id());
38
39     // Инициализируем генератор случайных чисел для случайных периодов сна
40     std::random_device rd;
41     std::default_random_engine engine{rd()};
42     std::uniform_int_distribution ints{0, 5000};
43
44     // Сохраняем задачи, чтобы дождаться их выполнения впоследствии; запись
45     // concurrencpp::result<void> означает, что каждая задача возвращает void
46     std::vector<concurrencpp::result<void>> results;
47
48     std::cout << "STARTING THREE CONCURRENCPP TASKS\n";
49
50     // Планируем три задачи
51     for (int i{1}; i < 4; ++i) {
52        std::chrono::milliseconds sleepTime{ints(engine)};
53        std::string name{fmt::format("Task {}", i)};
54
55        // Используем thread_pool_executor библиотеки concurrencpp для
56        // планирования вызова функции printTask с аргументами name и sleepTime
57        results.push_back(runtime.thread_pool_executor()->submit(
58           printTask, name, sleepTime));
59     }
60
61     std::cout << "\nALL TASKS STARTED\n";
62     std::cout << "\nWAITING FOR TASKS TO COMPLETE\n";
63
64     // Ждем завершения каждой задачи
65     for (auto& result : results) {
66        result.get(); // Ждем, когда каждая задача возвратит свой результат
67     }
68
69     std::cout << fmt::format("main's thread ID: {}\nMAIN ENDS\n", id());
70 }
```

```
main's thread ID: 20740
STARTING THREE CONCURRENCPP TASKS
ALL TASKS STARTED
```

Окончание ⇨

```
WAITING FOR TASKS TO COMPLETE
Task 3 (thread ID: 18960) going to sleep for 2683 ms
Task 1 (thread ID: 9840) going to sleep for 3856 ms
Task 2 (thread ID: 1700) going to sleep for 904 ms
Task 2 (thread ID: 1700) done sleeping
Task 3 (thread ID: 18960) done sleeping
Task 1 (thread ID: 9840) done sleeping
main's thread ID: 20740
MAIN ENDS
```

Илл. 18.2 | Планирование задач с помощью concurrencpp::runtime

Функция id

Функция `id` (строки 13–17) создает строковое представление уникального идентификатора текущего потока. Это представление понадобится нам, чтобы видеть, в каких потоках выполняются функция `main` и каждая из наших задач.

Функция printTask

В этом примере нам не приходится создавать потоки и управлять ими. Когда вы планируете выполнение задач с помощью объекта `concurrencpp::thread_pool_executor`, он создает и координирует потоки без вашего участия, используя свой пул потоков. В этом примере запланированы вызовы функции `printTask` (строки 21–31) и показано, что они выполняются в отдельных потоках. В строках 22–24 и 29–30 уникальный идентификатор текущего потока выводится на экран, чтобы подтвердить, что `printTask` вызывается из разных потоков.

Функция main

Функция `main` похожа на ту, которая показана на илл. 17.4, поэтому мы отметим только инструкции, связанные с библиотекой `concurrencpp`. В строке 35 создается объект `concurrencpp::runtime`, что дает возможность использовать функциональность этой библиотеки. В строке 37 выводится на экран идентификатор потока, в котором выполняется функция `main`.

Задачи и результаты

Задачи дают возможность разработчикам быть более продуктивными, «позволяя им уделять больше внимания бизнес-логике и меньше — низкоуровневым концепциям, таким как управление потоками и межпоточная синхронизация»[1]. Вы определяете свои задачи как функции (или лямбды, или функциональные объекты). Когда вы планируете задачу, `concurrencpp` создает для вас объект `concurrencpp::task` и возвращает объект `concurrencpp::result`, представляющий результат задачи, который может стать доступным не сразу, а только

[1] Дэвид Хайм (David Haim), «concurrencpp overview», Был доступен: 15 сентября 2023 г. https://github.com/David-Haim/concurrencpp#concurrencpp-overview.

когда-либо в будущем. Если функция задачи возвращает результат, вы получаете доступ к нему через объект `concurrencpp::result`, как мы покажем на илл. 18.3. Если функция возвращает `void`, как это делает `printTask`, вы можете использовать объект `concurrencpp::result<void>`, чтобы дождаться завершения задачи, как в случае с присоединением потока. В строке 46 создается вектор объектов `concurrencpp::result<void>`. Мы используем его для хранения результатов задач, поэтому можем ждать в функции `main`, когда эти задачи будут завершены.

concurrencpp::thread_pool_executor

В таких языках программирования, как Java, Go, Python и Kotlin, рекомендуется реализовывать конкурентность через **исполнители (executors)** вместо того, чтобы вручную создавать и координировать потоки. В данном примере для планирования каждой задачи используется исполнитель типа `concurrencpp::thread_pool_executor` (строки 57–58). Функция `thread_pool_executor` объекта `runtime` возвращает умный указатель `shared_ptr`. Во всех частях вашей программы умные указатели `shared_ptr` используются для взаимодействия с одним объектом исполнителя этого типа. Функция `submit` исполнителя планирует выполнение задачи. Она получает переменное количество аргументов:

- Первым аргументом функции `submit` является функция, определяющая задачу, которую надо выполнить, — в данном случае это `printTask`.
- Все последующие аргументы функция `submit` передает функции, указанной в ее первом аргументе, — в данном случае `submit` передает функции `printTask` аргументы `name` и `sleepTime`.

Каждый вызов функции `submit` создает объект задачи и возвращает результат. Функция `printTask` возвращает `void`, поэтому `submit` в этом примере возвращает тип `concurrencpp::result<void>`.

Ожидание завершения выполнения задач

Вызов функции `get` каждого объекта `result` (строки 65–67) ожидает результата соответствующей задачи — это напоминает присоединение потока. Каждая задача в этом примере возвращает `void`, поэтому вызов функции `get` просто заставляет функцию `main` ждать завершения задачи. Если бы `printTask` вернула значение, то функция `get` вернула бы это значение.

Исполнители библиотеки concurrencpp

В документации библиотеки `concurrencpp` перечислены имеющиеся в ней исполнители[1], в том числе:

[1] Дэвид Хайм (David Haim). «concurrencpp — Executors». Был доступен: 15 сентября 2023 г. https://github.com/David-Haim/concurrencpp#executors.

- `thread_executor` — для каждой задачи исполнитель `thread_executor` запускает отдельный поток, который не используется повторно после ее завершения. Согласно документации `concurrencpp`, исполнитель `thread_executor` «хорошо подходит для долго выполняющихся задач, таких как объекты, выполняющие рабочий цикл, или длительные блокирующие операции».
- `thread_pool_executor` (см. илл. 18.2 и 18.3) планирует задачи, используя пул потоков. «Подходит для коротких задач, которые не блокируют процессор. В приложениях рекомендуется использовать этот исполнитель в качестве исполнителя по умолчанию для неблокирующих задач».
- `background_executor` — «многопоточный исполнитель, использующий большой пул потоков. Подходит для запуска коротких блокирующих задач, таких как ввод-вывод файлов и запросы к базам данных».
- `worker_thread_executor` — «однопоточный исполнитель, обслуживающий единственную очередь задач. Подходит, когда приложению нужен выделенный поток, выполняющий много взаимосвязанных задач».
- `inline_executor` — «в основном нужен для того, чтобы переопределять поведение других исполнителей. Добавление задачи в очередь эквивалентно ее `inline`-вызову».

Задачи не обязаны выполняться в разных потоках

Исполнитель `concurrencpp::inline_executor` планирует выполнение задач в вызывающем потоке. Чтобы получить показанный ниже результат, мы заменили `thread_pool_executor` (илл. 18.2, строка 57) на `inline_executor`, который выполняет задачи в вызывающем потоке (в данном случае в потоке функции `main`). Каждый раз, когда вы запускаете эту программу с исполнителем `inline_executor`, задачи последовательно выполняются в одном потоке с функцией `main` в том порядке, в каком вы их запланировали. Этот пример выходных данных показывает, что у всех трех задач тот же идентификатор потока, что и у функции `main`. Когда мы планируем выполнение каждой задачи, она сразу же переходит в спящий режим, затем просыпается и завершает работу перед запуском следующей задачи.

```
main's thread ID: 7854

STARTING THREE CONCURRENCPP TASKS
Task 1 (ID: 7854) going to sleep for 2000 ms
Task 1 (ID: 7854) done sleeping
Task 2 (ID: 7854) going to sleep for 4432 ms
Task 2 (ID: 7854) done sleeping
Task 3 (ID: 7854) going to sleep for 3688 ms
Task 3 (ID: 7854) done sleeping

ALL TASKS STARTED
```

```
WAITING FOR TASKS TO COMPLETE

main's thread ID: 7854
MAIN ENDS
```

18.6. Создание корутины, содержащей co_await и co_return

Теперь давайте используем библиотеку `concurrencpp`, а также ключевые слова `co_await` и `co_return` для реализации корутины, выполняющей длительную задачу: сортировку вектора (контейнера `vector`), содержащего 100 миллионов чисел `int`.

Обзор этой программы

Мы будем применять исполнитель `thread_pool_executor` библиотеки `concurrencpp` для конкурентной работы двух задач, каждая из которых сортирует половину вектора. После завершения обеих задач мы объединим обе отсортированные половины вектора алгоритмом `inplace_merge` стандартной библиотеки (раздел 14.4.9). Программа содержит следующие функции:

- Функция `id` (строки 14–18) преобразует уникальный идентификатор потока в строку.
- Корутина `sortCoroutine` (строки 21–78) использует `concurrencpp` для запуска двух задач, каждая из которых сортирует половину вектора.
- Функция `main` (строки 80–107) создает вектор из 100 миллионов элементов и вызывает корутину `sortCoroutine` для его сортировки.

Мы разделили эту программу на части для удобства обсуждения (илл. 18.3). В конце показан пример выходных данных.

Директивы #include и функция id

В строках 3–11 включены все необходимые заголовки, а в строках 14–18 определена функция `id`. Необходимые средства поддержки корутин предоставляет библиотека `concurrencpp` (строка 3).

```
 1 // fig18_03.cpp
 2 // Реализация корутины, содержащей co_await и co_return
 3 #include <concurrencpp/concurrencpp.h>
 4 #include <fmt/format.h>
 5 #include <iostream>
 6 #include <memory> // Для shared_ptr
 7 #include <random>
 8 #include <sstream>
 9 #include <string>
10 #include <thread>
11 #include <vector>
```

Окончание ⇨

```
12
13 // Получаем идентификатор текущего потока в виде строки
14 std::string id() {
15     std::ostringstream out;
16     out << std::this_thread::get_id();
17     return out.str();
18 }
19
```

Илл. 18.3 | Реализация корутины, содержащей co_await и co_return

Корутина sortCoroutine запускает две задачи

В строках 21–78 определена корутина `sortCoroutine`, получающая в качестве аргументов:

- исполнитель `std::shared_ptr<concurrencpp::thread_pool_executor>`, применяемый для планирования задач;
- константную ссылку на `vector<int>`, который надо отсортировать.

```
20 // Корутина, сортирующая vector<int> в двух задачах
21 concurrencpp::result<void> sortCoroutine(
22     std::shared_ptr<concurrencpp::thread_pool_executor> executor,
23     std::vector<int>& values) {
24
25     std::cout << fmt::format("Thread {}: sortCoroutine started\n\n", id());
26
27 // Лямбда, сортирующая часть вектора
28     auto sortTask(
29         [&](auto begin, auto end) {
30             std::cout << fmt::format(
31                 "Thread {}: Sorting {} elements\n", id(), end - begin);
32             std::sort(begin, end);
33             std::cout << fmt::format("Thread {}: Finished sorting\n", id());
34         }
35     );
36
37     // Для хранения результатов задач
38     std::vector<concurrencpp::result<void>> results;
39
40     size_t middle{values.size() / 2}; // Индекс среднего элемента
41
42     std::cout << fmt::format(
43         "Thread {}: sortCoroutine starting first half sortTask\n", id());
44
45     // Используем thread_pool_executor для планирования
46     // вызова sortTask, сортирующего первую половину значений
47     results.push_back(
48         executor->submit(
49             [&]() {sortTask(values.begin(), values.begin() + middle);}
50         )
```

```
51    );
52
53    std::cout << fmt::format(
54       "Thread {}: sortCoroutine starting second half sortTask\n", id());
55
56    // Используем thread_pool_executor для планирования
57    // вызова sortTask, сортирующего вторую половину значений
58    results.push_back(
59       executor->submit(
60          [&]() {sortTask(values.begin() + middle, values.end());}
61       )
62    );
63
64    // Приостанавливаем корутину в ожидании завершения всех вызовов sortTask
65    std::cout << fmt::format("\nThread {}: {}\n", id(),
66       "sortCoroutine co_awaiting sortTask completion");
67    co_await concurrencpp::when_all(
68       executor, results.begin(), results.end());
69
70    // Объединяем две отсортированные половины вектора
71    std::cout << fmt::format(
72       "\nThread {}: sortCoroutine merging results\n", id());
73    std::inplace_merge(
74       values.begin(), values.begin() + middle, values.end());
75
76    std::cout << fmt::format("Thread {}: sortCoroutine done\n", id());
77    co_return; // Завершаем корутину и возвращаем управление в точку вызова
78 }
79
```

Корутина `sortCoroutine` работает так:

- В строке 25 выводится сообщение, содержащее идентификатор потока, в котором выполняется корутина `sortCoroutine`, и указывающее, что она запущена. Идентификатор потока подтверждает, что `sortCoroutine` выполняется в том же потоке, что и `main`.
- Корутина `sortCoroutine` запускает две задачи `concurrencpp`, выполняющие лямбду `sortTask` (строки 28–35). Лямбда получает итераторы произвольного доступа, представляющие начало и конец обычного диапазона, который надо отсортировать. В строках 30–31 выводится на экран сообщение, содержащее идентификатор потока, в котором выполняется текущий вызов `sortTask`, и количество элементов, которые он сортирует. В строке 32 вызывается алгоритм `std::sort` для сортировки указанной части вектора. Когда сортировка завершается, в строке 33 выводится на экран сообщение, содержащее идентификатор потока и указывающее, что задача сортировки завершена.
- Лямбда `sortTask` не возвращает значения, поэтому функция `submit` каждого исполнителя `thread_pool_executor` будет возвращать `concurrencpp::result<void>` для каждой задачи. Мы будем хранить эти результаты в векторе `results` (определенном в строке 38).

- В строке 40 определяется средний элемент сортируемого вектора, который мы будем использовать для разделения вектора пополам.
- В строках 42–43 выводится сообщение о запуске `sortTask` для первой половины вектора, содержащее идентификатор потока. Затем в строках 47–51 функция `submit` потока `thread_pool_executor` используется для создания задачи, которая вызывает `sortTask` для сортировки элементов от `values.begin()` до среднего элемента, но не включая его. Мы храним объект результата, возвращенный функцией `submit`, в векторе `results`. В строках 53–62 эти шаги повторяются для запуска второй задачи, которая сортирует элементы `values` от среднего элемента до конца.
- В строках 65–66 выводится сообщение о том, что корутина `sortCoroutine` ожидает результата `sortTask`, содержащего идентификатор потока, в котором она выполняется. В строках 67–68 функция `when_all` библиотеки `concurrencpp` используется для ожидания всех результатов `sortTask`. Скоро мы обсудим эту инструкцию подробнее.
- В строках 71–72 выводится сообщение о том, что корутина `sortCoroutine` объединяет результаты, содержащее идентификатор потока, в котором она выполняется. Затем в строках 73–74 алгоритм `inplace_merge` стандартной библиотеки объединяет две отсортированные части вектора.
- В строке 76 выводится сообщение о завершении корутины `sortCoroutine`, содержащее идентификатор потока, в котором она выполняется. Затем в строке 77 инструкция `co_return` завершает работу корутины и возвращает управление в `main`. Поскольку `sortCoroutine` не возвращает никакого значения, `concurrencpp` возвращает в точку вызова объект `concurrencpp::result<void>`. Вы увидите, что `main` использует функцию `get` этого объекта, чтобы дождаться завершения выполнения программы.

Функция when_all в сочетании с выражением co_await

Функция `when_all` библиотеки `concurrencpp` позволяет программе ожидать завершения набора задач. Вероятно, мы увидим такую функцию в стандартной библиотеке C++23 или последующих версий. Функция `when_all` получает в качестве аргументов:

- исполнитель, который нужен для возобновления корутины после завершения всех задач;
- итераторы, представляющие обычный диапазон объектов `concurrencpp::result`, — в данном случае это все элементы вектора `results`.

Инструкция в строках 67–68 ожидает результата вызова функции `when_all`. Выражение `co_await` состоит из оператора `co_await`, за которым следует выражение, возвращающее ожидаемый объект. Обычно это объект класса библиотеки корутин. Объект `concurrencpp::result` удовлетворяет требованиям стандарта C++ к операнду оператора `co_await`, так же как и объект, возвращаемый функцией `when_all`,

который представляет собой объект `concurrencpp::lazy_result`, содержащий кортеж с результатами всех задач.

Проверка необходимости приостановки корутины

В этот момент корутина `sortCoroutine` определяет, надо ли приостановить выполнение. Для этого она вызывает функцию `await_ready` операнда `co_await`, чтобы проверить, доступен ли уже результат задачи. Если да, то обе задачи `sortTask` уже завершены, поэтому корутина продолжает выполнение со следующей по порядку инструкции. В противном случае корутина приостанавливается до тех пор, пока не завершатся асинхронные задачи в аргументах `when_all`. Это позволяет вызывающей стороне (`main`) выполнять другую работу, которая не зависит от результатов асинхронных задач.

Как видно из примера выходных данных, если корутина приостанавливается, функция `main` продолжает выполняться, показывая это выводом на экран соответствующей строки текста. Затем функция `main` ожидает, когда корутина `sortCoroutine` завершится и окончательно вернет управление. Во время этого ожидания функция `main` может продолжать другую работу.

Возобновление корутины

После завершения всех асинхронных задач, которых дожидалась корутина `sortCoroutine`, она продолжает выполняться со следующей инструкции после выражения `co_await`. В этом примере `sortCoroutine` объединяет обе отсортированные части вектора, после чего инструкцией `co_return` возвращает управление в `main`.

Функция when_any

В библиотеке `concurrencpp` также есть функция `when_any`, которая позволяет программе ожидать завершения любой из двух или более задач. Такую функцию мы тоже можем увидеть в стандартной библиотеке C++23 или последующих версий. Она нужна, например, при загрузке нескольких больших файлов — по одному на каждую задачу. Хотя в конечном итоге вам могут понадобиться все результаты, вы хотели бы начать обработку сразу после завершения первой загрузки. Затем вы можете снова вызвать `when_any` для оставшихся задач, которые еще выполняются. Функция `when_any` обычно используется в цикле, который продолжается до тех пор, пока не завершится последняя из нескольких задач.

Функция main

Функция `main` (строки 80–107) создает вектор, содержащий 100 миллионов случайных чисел `int`, затем вызывает корутину `sortCoroutine` для его сортировки, ожидает завершения корутины и подтверждает, что вектор отсортирован.

```
80 int main() {
81     concurrencpp::runtime runtime; // Создаем объект runtime
82     auto executor{runtime.thread_pool_executor()}; // Получаем исполнитель
```

```
83
84    // Инициализируем генератор случайных чисел
85    std::random_device rd;
86    std::default_random_engine engine{rd()};
87    std::uniform_int_distribution ints;
88
89    std::cout << fmt::format(
90       "Thread {}: main creating vector of random ints\n", id());
91    std::vector<int> values(100'000'000);
92    std::ranges::generate(values, [&]() {return ints(engine);});
93
94    std::cout << fmt::format(
95       "Thread {}: main starting sortCoroutine\n", id());
96    auto result{sortCoroutine(executor, values)};
97
98    std::cout << fmt::format("\nThread {}: {}\n", id(),
99       "main resumed. Waiting for sortCoroutine to complete.");
100   result.get(); // Ждем завершения корутины sortCoroutine
101
102   std::cout << fmt::format(
103      "\nThread {}: main confirming that vector is sorted\n", id());
104   bool sorted{std::ranges::is_sorted(values)};
105   std::cout << fmt::format("Thread {}: values is{} sorted\n",
106      id(), sorted ? "" : " not");
107 }
```

Функция `main` работает так:

- В строках 81–82 создаются объект `concurrencpp::runtime` и его исполнитель `concurrencpp::thread_pool_executor`.
- В строках 85–87 настраиваются параметры генератора случайных чисел для заполнения ими вектора `vector<int>`.
- В строках 89–92 создается вектор `vector<int>` размером 100 миллионов элементов, который затем заполняется случайными числами `int` с помощью алгоритма `std::ranges::generate`.
- В строках 94–95 выводится сообщение о том, что поток `main` выполняется и собирается вызвать корутину `sortCoroutine`.
- В строке 96 вызывается корутина `sortCoroutine`, которой передаются исполнитель и вектор. Она запускает асинхронные задачи сортировки и возвращает объект типа `concurrencpp::result<void>`. Вызов корутины `sortCoroutine` выполняется в потоке `main` для запуска асинхронных задач. Когда выполняемая корутина `sortCoroutine` достигает строки с выражением `co_await` для этих задач, она приостанавливается, если результаты задач еще не получены, а в противном случае просто завершается и возвращает управление.
- В данном случае асинхронные задачи не смогут завершиться быстро, поскольку каждой задаче, которую запускает корутина `sortCoroutine`, нужно время для сортировки 50 миллионов элементов. Поэтому корутина приоста-

навливается и возвращает управление функции `main`. В этот момент в строках 98–99 выводится сообщение о том, что функция `main` снова выполняется. С этого момента функция `main` потенциально могла бы продолжить какую-либо другую работу.

- В строке 100 вызывается метод `get` объекта `result`, чтобы заблокировать выполнение `main` до тех пор, пока не завершатся асинхронные задачи корутины `sortCoroutine`.
- Сразу после завершения задач в строках 102–103 выводится сообщение о том, что функция `main` подтверждает выполненную сортировку вектора `values`. Затем в строке 104 вызывается алгоритм `std::ranges::is_sorted`, которому в качестве аргумента передается вектор `values`. Этот алгоритм возвращает `true`, если его аргумент отсортирован; в противном случае он возвращает `false`. В строках 105–106 выводится сообщение об итоге сортировки.

Пример выходных данных

Давайте детально обсудим наш пример выходных данных. Сначала они приведены целиком, а потом по частям для удобства обсуждения.

```
Thread 17276: main creating vector of random ints
Thread 17276: main starting sortCoroutine
Thread 17276: sortCoroutine started

Thread 17276: sortCoroutine starting first half sortTask
Thread 17276: sortCoroutine starting second half sortTask

Thread 17276: sortCoroutine co_awaiting sortTask completion

Thread 17276: main resumed. Waiting for sortCoroutine to complete.
Thread 17144: Sorting 50000000 elements
Thread 6252: Sorting 50000000 elements
Thread 6252: Finished sorting
Thread 17144: Finished sorting

Thread 6252: sortCoroutine merging results
Thread 6252: sortCoroutine done

Thread 17276: main confirming that vector is sorted
Thread 17276: values is sorted
```

Сначала функция `main` выводит на экран идентификатор своего потока и создает вектор случайных целых чисел:

```
Thread 17276: main creating vector of random ints
```

Далее `main` снова выводит на экран идентификатор своего потока и вызывает корутину `sortCoroutine`:

```
Thread 17276: main starting sortCoroutine
```

Корутина `sortCoroutine` выводит на экран идентификатор своего потока и сообщает, что она запущена:

```
Thread 17276: sortCoroutine started
```

Обратите внимание, что `sortCoroutine` выполняется в одном потоке с функцией `main`, поэтому выводит тот же идентификатор потока.

Далее корутина `sortCoroutine` запускает две задачи для сортировки вектора:

```
Thread 17276: sortCoroutine starting first half sortTask
Thread 17276: sortCoroutine starting second half sortTask
```

После запуска задач корутина `sortCoroutine` ждет их завершения:

```
Thread 17276: sortCoroutine co_awaiting sortTask completion
```

Поскольку задачи `sortTask` не завершены, корутина `sortCoroutine` приостанавливается и возвращает управление функции `main`. Мы специально создали большой вектор, чтобы сортировка не завершилась быстро. Это позволяет показать, что `main` продолжает выполняться, когда корутина приостанавливает выполнение:

```
Thread 17276: main resumed. Waiting for sortCoroutine to complete.
```

Функция `main` выводит это сообщение, затем вызывает функцию `get` объекта `result`, то есть блокируется до тех пор, пока `sortCoroutine` не завершится и не вернет управление.

Тем временем параллельные задачи `sortTask` начинают выполняться в других потоках, о чем свидетельствуют идентификаторы этих потоков:

```
Thread 17144: Sorting 50000000 elements
Thread 6252: Sorting 50000000 elements
```

Мы не можем предсказать относительную скорость асинхронных конкурентно выполняемых задач, поэтому не знаем, какая из них завершится первой. В данном случае задача, запущенная первой, завершилась последней, что видно по идентификаторам потоков:

```
Thread 6252: Finished sorting
Thread 17144: Finished sorting
```

Корутина `sortCoroutine` объединяет результаты двух задач, а затем сообщает, что все готово:

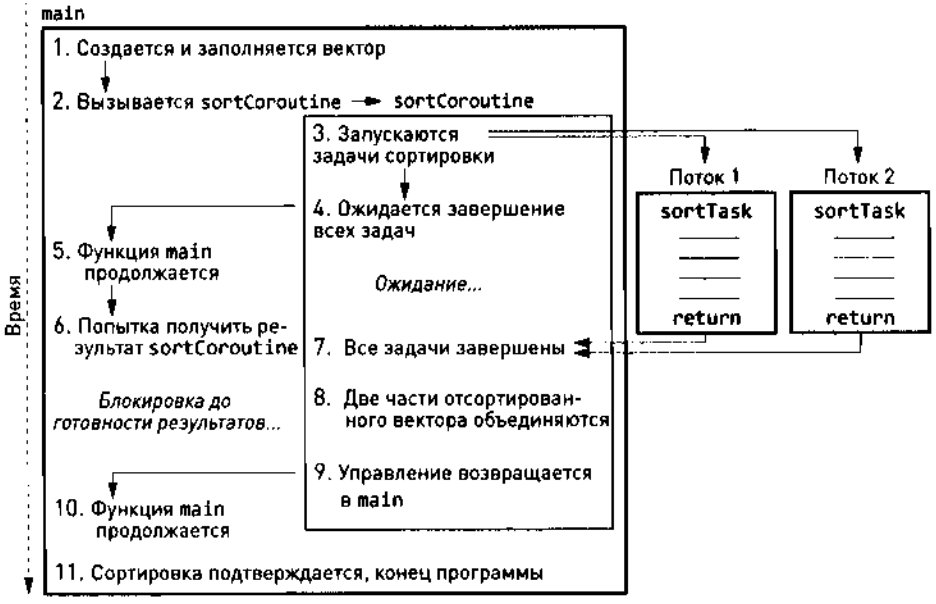
```
Thread 17144: sortCoroutine merging results
Thread 17144: sortCoroutine done
```

Как видите, исполнитель возобновил выполнение корутины в том же потоке, что и задача `sortTask`, завершившаяся последней. Теперь корутина окончательно возвращает управление функции `main`, которая продолжает выполнение, подтверждает, что вектор отсортирован, и завершается:

```
Thread 17276: main confirming that vector is sorted
Thread 17276: values is sorted
```

Поток управления для корутины, содержащей co_await и co_return

Основной поток управления (flow of control) этой программы показан на диаграмме. Номера на диаграмме соответствуют номерам в ее описании:



1. Функция `main` создает и заполняет вектор случайных целых чисел.
2. Функция `main` вызывает корутину `sortCoroutine`.
3. Корутина `sortCoroutine` запускает две параллельные задачи, каждая из которых вызывает лямбду `sortTask` для половины вектора. Эти задачи начинают выполняться параллельно.
4. Корутина `sortCoroutine` в ожидании результатов всех задач приостанавливается и возвращает управление функции `main`.
5. Функция `main` продолжает выполнение.
6. Функция `main` пытается получить результат корутины `sortCoroutine`. Это блокирует продолжение функции `main` до тех пор, пока `sortCoroutine` не завершит выполнение и не вернет управление функции `main`.
7. Через некоторое время задачи `sortTask` завершаются, после чего корутина `sortCoroutine` возобновляет выполнение.
8. Корутина `sortCoroutine` объединяет отсортированные части вектора.
9. Корутина `sortCoroutine` возвращает управление функции `main` и завершается.
10. Функция `main` возобновляет выполнение.
11. Функция `main` подтверждает, что вектор отсортирован, и завершает работу.

18.7. Низкоуровневые концепции корутин

Вы уже видели, что с такими библиотеками поддержки корутин, как `generator` и `concurrencpp`, создавать корутины очень просто. Но для тех, кто хочет попробовать программировать корутины без этих библиотек (напомним, называемых «экспериментальными»), мы приводим в этом разделе обзор нескольких необходимых для этого низкоуровневых концепций. Для дальнейшего изучения мы приводим в сносках ключевые статьи и видеоуроки с примерами кода[1, 2, 3, 4, 5, 6].

Этот раздел также укрепит ваше понимание высокоуровневого, основанного на библиотеках подхода, который мы использовали в предыдущих разделах. Подробности по темам, обсуждаемым в этом разделе, можно найти в разделах 7.6.2.3, 7.6.17, 8.7.4, 9.5.4 и 17.12 стандарта C++[7], а также на сайте cppreference.com[8, 9].

Ограничения корутин

Согласно стандарту C++20, следующие виды функций не могут быть корутинами[10]:

- функция `main`;
- конструкторы;
- деструкторы;
- `constexpr`-функции и `consteval`-функции;
- функции, объявленные с подставляемыми типами (`auto` или концепты C++20);
- функции с переменным числом аргументов.

---

[1] Мариус Банчила (Marius Bancila), «Modern C++ Programming Cookbook», глава 12, с. 681–700. Packet, 2020 г.

[2] Саймон Тот (Simon Toth), «C++20 Coroutines: Complete Guide», 26 сентября 2021 г. Был доступен: 15 сентября 2023 г. https://itnext.io/c-20-coroutines-complete-guide-7c3fc08db89d.

[3] Райнер Гримм (Rainer Grimm), «40 Years of Evolution from Functions to Coroutines», 25 сентября 2020 г. Был доступен: 15 сентября 2023 г. https://www.youtube.com/watch?v=jd6P9X8l2bY.

[4] Джефф Томас (Jeff Thomas), «Exploring Coroutines», 7 апреля 2021. Был доступен: 15 сентября 2023 г. https://blog.coffeetocode.com/2021/04/exploring-coroutines.

[5] Panicsoftware, «Your First Coroutine», 6 марта 2019 г. Был доступен: 15 сентября 2023 г. https://blog.panicsoftware.com/your-first-coroutine.

[6] Panicsoftware, «co_awaiting Coroutines», 12 апреля 2019 г. Был доступен: 15 сентября 2023 г. https://blog.panicsoftware.com/co_awaiting-coroutines.

[7] Стандарт C++. Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861.

[8] «Coroutines (C++20)». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/language/coroutines.

[9] «Standard Library Header <coroutine>». Был доступен: 15 сентября 2023 г. https://en.cppreference.com/w/cpp/header/coroutine.

[10] Стандарт C++, разделы 6.9.3.1, 9.2.5, 9.2.8.5, 11.4.4, 11.4.6. Был доступен: 15 сентября 2023 г. https://timsong-cpp.github.io/cppwp/n4861.

Объект promise

Каждая корутина имеет объект `promise`[1], который создается при ее первом вызове. Он используется корутиной для того, чтобы:

- возвращать результат в точку вызова;
- возвращать исключение в точку вызова.

Объект `promise` является частью состояния корутины. Он предоставляет несколько функций:

- `get_return_object` — когда корутина начинает выполняться, она вызывает эту функцию для объекта `promise`, чтобы получить объект результата корутины, который будет возвращен в точку вызова корутины, когда она впервые приостановится. Вызывающая сторона использует объект результата для доступа к результату корутины, когда он становится доступным.
- `initial_suspend` — когда корутина начинает выполняться, она вызывает эту функцию для объекта `promise`, чтобы определить, следует ли немедленно приостановиться (для ленивых корутин, таких как наша `fibonacciGenerator`, илл. 18.1) или продолжить выполнение (как наша корутина `sortCoroutine`, илл. 18.3). Заголовок `<coroutine>` предоставляет типы `suspend_always` и `suspend_never` для поддержки этих двух вариантов.
- `unhandled_exception` — корутина вызывает эту функцию для объекта `promise`, если необработанное исключение приводит к завершению работы корутины.
- `return_void` — корутина вызывает эту функцию для объекта `promise`, когда поток управления достигает закрывающей скобки корутины, или инструкции `co_return` без выражения, или инструкции `co_return` с выражением, вычисляемым как `void`. Объект `promise` может определять либо `return_void`, либо `return_value`, но не обе эти функции одновременно.
- `return_value` — корутина вызывает эту функцию, когда инструкция `co_return` содержит непустое выражение. Объект `promise` может определять либо `return_value`, либо `return_void`, но не обе эти функции одновременно.
- `final_suspend` — после вызова функции `return_void` или `return_value` корутина вызывает эту функцию, чтобы вернуть управление и, возможно, результат, в точку вызова.
- `yield_value` — когда выполняется инструкция `co_yield`, корутина-генератор вызывает эту функцию, чтобы вернуть значение в точку вызова.

---

[1] Льюис Бейкер (Lewis Baker), «C++ Coroutines: Understanding the Promise Type», 5 сентября 2018 г. Был доступен: 15 сентября 2023 г. https://lewissbaker.github.io/2018/09/05/understandingthe-promise-type.

Состояние корутины

Состояние корутины (coroutine state), иногда называемое **кадром корутины (coroutine frame)**, содержит следующие данные:

- копии параметров корутины;
- объект `promise`;
- представление точки приостановки (чтобы корутина знала, откуда надо возобновить выполнение);
- локальные переменные, находящиеся в области видимости в точке приостановки.

В нашем примере генератора (илл. 18.1) точка приостановки находится там, где инструкция `co_yield` приостанавливает выполнение корутины и возвращает значение в точку вызова. В нашем примере сортировки (илл. 18.3) точка приостановки находится там, где мы ожидаем (`co_await`) результата вызова функции `concurrencpp::when_all`.

Шаблон класса coroutine_handle

Библиотеки поддержки генераторов используют шаблон класса `coroutine_handle` (заголовок `<coroutine>`) для взаимодействия с запущенной или приостановленной корутиной. В частности, `coroutine_handle` позволяет:

- создавать `coroutine_handle` из объекта `promise`;
- проверять, завершилось ли выполнение корутины;
- возобновлять приостановленную корутину;
- уничтожать состояние корутины;
- получать объект `promise` корутины.

Ожидаемые объекты

Операндом оператора `co_await` должен быть ожидаемый объект, такой как `concurrencpp::result` (илл. 17.21). Такие объекты должны предоставлять несколько функций:

- `await_ready` — когда вы ожидаете (`co_await`) ожидаемый объект, прежде всего вызывается эта функция, чтобы выяснить, готов ли уже результат. Если да, то корутина продолжает выполнение.
- `await_suspend` — если результат еще не готов, вызывается эта функция, чтобы выяснить, следует ли приостановить выполнение корутины и вернуть управление в точку вызова. Если эта функция возвращает `coroutine_handle` для другой приостановленной корутины, та корутина возобновляет выполнение.
- `await_resume` — когда корутина возобновляет выполнение, вызывается эта функция, чтобы вернуть результат выражения `co_await`.

18.8. Развитие корутин в C++23

В разделе 18.5 демонстрировались исполнители библиотеки `concurrencpp`, предназначенные для запуска задач и для управления потоками. Вероятно, аналогичные исполнители и другие средства поддержки корутин будут включены в стандартную библиотеку C++23. Кроме того, экспериментальные реализации исполнителей есть в нескольких других сторонних библиотеках. Например, библиотека `libunifex` («унифицированные исполнители»)[1], представляющая собой прототип реализации исполнителей и других средств асинхронного программирования, рассматривается на предмет стандартизации. Более подробную информацию вы можете найти в документах Комитета по стандартам C++:

- «std::execution»[2];
- «A Unified Executors Proposal for C++»[3];
- «Dependent Execution for a Unified Executors Proposal for C++»[4].

18.9. Итоги

В этой главе мы представили подробный высокоуровневый обзор корутин — последнего из «большой четверки» новшеств C++20 (диапазоны, концепты, модули и корутины). Ограниченная поддержка корутин в стандартной библиотеке C++20 рассчитана на разработчиков, создающих высокоуровневые библиотеки поддержки корутин для C++23 и последующих версий. Мы применили этот высокоуровневый, основанный на библиотеках подход, чтобы легко создавать корутины и программировать сложные конкурентные задачи, сохраняя простой последовательный стиль кодирования. Мы использовали ключевое слово `co_yield` и библиотеку `generator` для реализации ленивой корутины-генератора. Мы рассказали об исполнителях, задачах и результатах библиотеки `concurrencpp`, используя их для выполнения задач и для создания корутины с ключевыми словами `co_await` и `co_return`.

Спасибо, что вы прочитали книгу «C++20 для программистов». Мы надеемся, что она вам понравилась и вы нашли ее интересной и познавательной. Больше всего мы надеемся, что вы почувствуете в себе силы применить изученные технологии для решения задач, с которыми встретитесь в своей карьере.

---

[1] «libunifex». Был доступен: 15 сентября 2023 г. https://github.com/facebookexperimental/libunifex.

[2] Михаль Доминьяк (Michal Dominiak), Льюис Бейкер (Lewis Baker), Ли Хоус (Lee Howes), Кирк Шуп (Kirk Shoop), Майкл Гарланд (Michael Garland), Эрик Ниблер (Eric Niebler) и Брайс Адельштейн Лельбах (Bryce Adelstein Lelbach), «std::execution», 4 октября 2021. Был доступен: 15 сентября 2023 г. https://wg21.link/p2300.

[3] Джаред Хоберок (Jared Hoberock), Майкл Гарланд, Крис Колхофф (Chris Kohlhoff), Крис Майсен (Chris Mysen), Картер Эдвардс (Carter Edwards), Гордон Браун (Gordon Brown), Дейзи Холлман (Daisy Hollman), Ли Хоус, Кирк Шуп, Льюис Бейкер и Эрик Ниблер, «A Unified Executors Proposal for C++», 15 сентября 2020 г. Был доступен: 15 сентября 2023 г. http://wg21.link/p0443.

[4] Джаред Хоберок, Майкл Гарланд, Крис Колхофф, Крис Майсен, Картер Эдвардс и Гордон Браун, «Dependent Execution for a Unified Executors Proposal for C++», 8 октября 2018 г. Был доступен: 15 сентября 2023 г. https://wg21.link/p1244.

Приложение А

Приоритеты и группировка операторов

Операторы показаны в порядке убывания приоритетов сверху вниз.

Оператор	Операция	Группировка (ассоциативность)
`::`	разрешение области видимости	слева направо
`()`	группирующие скобки	
`()`	вызов функции	слева направо
`[]`	индекс массива	
`.`	выбор элемента через объект	
`->`	выбор элемента через указатель	
`++`	унарный постфиксный инкремент	
`--`	унарный постфиксный декремент	
`typeid`	динамическая информация о типе	
`dynamic_cast<`*тип*`>`	динамическое приведение типа	
`static_cast<`*тип*`>`	статическое приведение типа	
`reinterpret_cast<`*тип*`>`	приведение типа для нестандартных операций	
`const_cast<`*тип*`>`	отмена квалификатора `const`	
`++`	унарный префиксный инкремент	справа налево
`--`	унарный префиксный декремент	
`+`	унарный плюс	
`-`	унарный минус	
`!`	унарное логическое отрицание	
`~`	побитовое отрицание	
`sizeof`	определение размера в байтах	
`&`	взятие адреса	
`*`	разыменование	

Оператор	Операция	Группировка (ассоциативность)
`new`	динамическое выделение памяти	
`new[]`	динамическое выделение памяти массива	
`delete`	динамическое освобождение памяти	
`delete[]`	динамическое освобождение памяти массива	
`co_await`	приостановка корутины	
(*тип*)	унарное приведение типа в стиле C	
`.*`	указатель на элемент через объект	слева направо
`->*`	указатель на элемент через указатель	
`*`	умножение	слева направо
`/`	деление	
`%`	остаток от деления	
`+`	сложение	слева направо
`-`	вычитание	
`<<`	побитовый сдвиг влево	слева направо
`>>`	побитовый сдвиг вправо	
`<=>`	трехстороннее сравнение	слева направо
`<`	меньше	слева направо
`<=`	меньше или равно	
`>`	больше	
`>=`	больше или равно	
`==`	равно	слева направо
`!=`	не равно	
`&`	побитовое И	слева направо

Оператор	Операция	Группировка (ассоциативность)
^	побитовое исключающее ИЛИ	слева направо
|	побитовое ИЛИ	слева направо
&&	логическое И	слева направо
||	логическое ИЛИ	слева направо
?:	тернарное условие	справа налево
=	присваивание	
+=	сложение с присваиванием	
-=	вычитание с присваиванием	
*=	умножение с присваиванием	
/=	деление с присваиванием	
%=	остаток от деления с присваиванием	
&=	побитовое И с присваиванием	
^=	побитовое исключающее ИЛИ с присваиванием	
|=	побитовое ИЛИ с присваиванием	
<<=	побитовый сдвиг влево с присваиванием	
>>=	побитовый сдвиг вправо с присваиванием	
co_yield	приостановка корутины и возврат значения	
throw	выдача исключения	
,	запятая	слева направо

Приложение Б

Набор символов

Набор символов ASCII

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

Цифры в левой части таблицы — это первые цифры десятичных чисел в диапазоне от 0 до 127, которые являются кодами символов ASCII, а цифры в верхней части таблицы — последние цифры этих чисел. Например, десятичное число 70 — это код символа «F», а десятичное число 38 — это код символа «&».

Пол Дейтел, Харви Дейтел

C++20 для программистов

Перевел с английского М. Фатеев

Руководитель дивизиона	*Ю. Сергиенко*
Ведущий редактор	*Е. Строганова*
Литературный редактор	*В. Здобнов*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 04.2024. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 - Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ЛИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 27.02.24. Формат 70×100/16. Бумага офсетная. Усл. п. л. 85,140. Тираж 1200. Заказ 1481.

Отпечатано в АО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул. Полиграфистов, 1
Сайт: www.chpd.ru. E-mail: sales@chpd.ru
тел: 8(495) 107-02-68



для программистов

Метод готовых объектов

ОТЗЫВЫ

«"C++20 для программистов" формирует понимание современного C++, что необходимо каждому программисту в нынешней экосистеме разработки программного обеспечения. Блестящая и неповторимая манера изложения, в которой Дейтелы подают материал, полностью соответствует требованиям к созданию ПО промышленного уровня. Я настоятельно рекомендую эту книгу всем, кто желает освоить C++, особенно для работы в среде профессиональных программистов, где невозможно разобраться в идиомах и паттернах современного C++ без вдумчивого руководства, которое вы найдете в книге».

— Д-р Дэйзи Холлман, член комитета ISO по стандартам языка C++

«Это отличная книга, в которой авторам удалось представить огромный объем материала по такому масштабному языку, как C++20. Глубокий разбор C++, предназначенный для читателей, знакомых с тем, как все работает в других языках программирования».

— Артур О'Двайр, преподаватель C++, председатель CppCon's Back to Basics track, автор нескольких принятых усовершенствований C++17/20/23 и автор книги «Осваиваем C++17 STL»

Программисты с опытом работы на других языках высокого уровня смогут на практике освоить **современный C++** и «большую четверку» его новых возможностей: **диапазоны, концепты, модули и корутины**.

200+ практических примеров реального исходного кода позволят быстро овладеть идиомами современного C++, используя популярные компиляторы: **Visual C++®, GNU® g++, Apple® Xcode® и LLVM®/Clang.** Знание базы позволит перейти к контейнерам стандартной библиотеки C++ `array` и `vector`; **функциональному программированию с диапазонами и представлениями C++20**; строкам, файлам и регулярным выражениям; **объектно-ориентированному программированию** с классами, наследованием, динамическим и статическим полиморфизмом; перегрузке операторов, семантике копирования и перемещения, RAII и умным указателям; исключениям и ожидаемым в **C++23 контрактам**, контейнерам, итераторам и алгоритмам стандартной библиотеки; шаблонам, **концептам C++20** и **метапрограммированию; модулям C++20** и технологии разработки больших программ; **конкурентности, параллелизму, параллельным алгоритмам стандартной библиотеки C++17 и C++20 и корутинам C++20**.

Пол и **Харви Дейтелы** — основатели компании **Deitel & Associates, Inc.**, получившей международное признание благодаря курсам по обучению языкам программирования и корпоративным тренингам. Миллионы людей по всему миру пользуются созданными Дейтелами книгами и видеокурсами LiveLessons, тренингами в прямом эфире O'Reilly Online Learning, корпоративными тренингами, проводимыми во многих странах мира (очными и виртуальными), а также интерактивными мультимедийными курсами Revel™ с интегрированными лабораторными работами и оценкой результатов, овладевая основными языками программирования и платформами, включая C++, Java®, Python®, C, C#, созданием мобильных приложений для iOS и Android, а также интернет- и веб-программированием.

ПИТЕР®

WWW.PITER.COM
интернет-магазин

Заказ книг:
(812) 703-73-74
books@piter.com

PiterBooks
PiterForPeople
ThePiterBooks
Company/piter

ISBN:978-5-4461-2359-9

Pearson