Андерс Шау Кнаттен

ХОРОШО ЛИ ВЫ ЗНАЕТЕ

ДМК пресс
издательство

Андерс Шау Кнаттен

Хорошо ли вы знаете C++?

C++ Brain Teasers

Exercise Your Mind

Anders Schau Knatten

Хорошо ли вы знаете C++?

25 задач для разминки ума

Андерс Шау Кнаттен

Москва, 2025

УДК 004.438C++(075)
ББК 32.73.2я7
К53

Андерс Шау Кнаттен

К53 Хорошо ли вы знаете C++? / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2024. – 138 с.: ил.

ISBN 978-5-93700-351-5

Эта книга содержит 25 тщательно отобранных задач, которые призваны проверить, насколько хорошо вы понимаете современные версии языка. Каждая задача сопровождается ответом и объяснениями. Ожидается, что читатель знает основы языка C++ и владеет навыками программирования.

Книга будет полезна как опытным специалистам, желающим углубленно изучить C++, так и приступающим к освоению языка.

УДК 004.438C++(075)
ББК 32.73.2я7

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN (анг.) 979-8-88865-051-6
ISBN (рус.) 978-5-93700-351-5
Copyright © 2024 The Pragmatic Programmers, LLC.
© Оформление, издание, перевод, ДМК Пресс, 2024

https://t.me/it_boooks/2

Оглавление

Предисловие от издательства 7

Вступительное слово Ольве Маудала 8

Благодарности 10

Предисловие 11

Задача 1. Сколько градусов? 17

Задача 2. Теория струн 19

Задача 3. Хакнем планету 25

Задача 4. На пути к глобализации 29

Задача 5. Деструктивные отношения 33

Задача 6. Кто первый? 37

Задача 7. Все хорошее должно когда-то заканчиваться 41

Задача 8. Переместится или нет? 47

Задача 9. Подсчет копий 53

Задача 10. Странное присваивание 59

Задача 11. Когда наступила смерть? 63

Задача 12. Фальстарт 69

Задача 13. Постоянная борьба 73

Задача 14. Аристотелева сумма частей 77

Задача 15. Назад из будущего 81

Задача 16. Перегруженный контейнер 85

Задача 17. Строгий указ 89

Задача 18. Освобождаем помещение 93

Задача 19. Маленькая сумма 97

Задача 20. Монстры на марше 105

Задача 21. Измерение некоторых символов 109

Задача 22. Космический корабль-призрак 113

Задача 23. Доброе начало полдела откачало 119

Задача 24. Специальная теория струн 123

Задача 25. Слабо типизированная, сильно озадачивающая 129

Предметный указатель 132

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте *www.dmkpress.com*, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу *dmkpress@gmail.com*; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу *http://dmkpress.com/authors/publish_book/* или напишите в издательство по адресу *dmkpress@gmail.com*.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу *dmkpress@gmail.com*, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу *dmkpress@gmail.com*.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Вступительное слово Ольве Маудала

Впервые я встретил Андерса Шау Кнаттена на местной встрече программистов C++ в Осло. С тех пор прошло довольно много времени, мы подружились и вместе проводили время, сталкиваясь на разных конференциях. Андерс проводил презентации на таких посвященных C++ мероприятиях, как ACCU, NDC TechTown, CppCon, C++ on Sea, Meeting C++ и многих других. Посещая их, я всегда узнаю что-то новенькое. Андерс может часами говорить о том, как на самом деле ведут себя целые числа, о том, что происходит перед вызовом main(), или о том, как эффективно работать с отладчиком. Он также обладает удивительной способностью заметить что-то, переосмыслить, придать другую форму и вывести на следующий уровень. Именно это он и сделал, посетив два моих печально известных семинара C++ Pub Quiz. Очень скоро Андерс создал изумительный сайт cppquiz.org[1] (загляните, не поленитесь!). А теперь он проводит свои собственные, значительно улучшенные версии таких семинаров на конференциях. В этой книге Андерс продолжил развивать эту идею.

Я люблю книги, люблю C++ и люблю задачки. В книге «Хорошо ли вы знаете C++?» все это сошлось в одном месте. Я изучаю и пользуюсь C++ вот уже больше 30 лет, но уверен, что буду читать и перечитывать эту книгу снова и снова и каждый раз находить что-то новое. Если вы глубоко привязаны к C++, то тоже захотите прочитать ее. Книга станет для вас дорогой к более глубокому освоению этого чрезвычайно мощного и чарующего языка программирования. Она содержит 25 тщательно отобранных задач, которые призваны испытать, насколько хорошо вы понимаете современные версии языка. После того как вы попытаетесь решить задачу самостоятельно, Андерс представит глубокое и основательное объяснение причин, по

---

[1] *https://cppquiz.org*.

которым эта задача интересна, и способов научиться рассуждать о таких вещах.

Вы можете написать на C++ красивые программы, но, конечно, можно сочинить и полную лажу. Важно понимать, что такое C++, потому что он уходит корнями в языки BCPL, C, Simula 67 и Algol 68. C++ стал тем, чем он является, потому что был быстро принят сообществом и набрал миллионы активных пользователей уже на протяжении самых первых лет. C++ стал таким, каким мы его знаем, потому что он доверяет программисту и сосредоточен на решении реальных задач. Да, C++ сложен, быть может, даже слишком сложен, но это язык, на который нужно go-to (игра слов не случайна!) при работе во многих областях, как то: операционные системы, встраиваемые системы, высокопроизводительные вычисления, особо быстрые приложения, видеоигры и космические станции. Приведем высказывание Бьярна Страуструпа (создателя C++), которое все расставляет по местам:

> Есть только два вида языков: те, на которые жалуются, и те, которыми никто не пользуется.

Желаю вам всего наилучшего в путешествии к более глубокому пониманию C++. На этот раз вашим гидом будет Андерс. Получайте удовольствие!

Ольве Маудал,
докладчик на международных конференциях по C и C++,
преподаватель и организатор конференций.
Июнь 2024

Благодарности

Прежде всего хочу поблагодарить Ольве Маудала и Ассоциацию пользователей C и C++ (ACCU), которые побудили меня создать сайт CppQuiz.org[1] во время конференции ACCU 2013. Я также благодарен всем, кто помогал в этом предприятии, наполняя сайт материалами, участвуя в реализации и присылая извещения об ошибках.

Большое спасибо моему редактору, Сандре Уильямс, за бесценную помощь на протяжении всей работы. Без твоего руководства, поддержки и советов эта книга оказалась бы гораздо хуже. Я также признателен Маргарет Элдридж, которая обратилась ко мне от имени издательства, и Франсису Буонтемпо, который познакомил нас.

Мне посчастливилось получать помощь от лучших известных мне специалистов по C++. Спасибо вам, Даниэла Энгерт, Бьярн Фаллер, Ольве Маудал, Картхик Нишант, Том Шульц, Тина Ульбрих, Сергей Васильченко и Пётр Вирчиньский, за работу в качестве технических редакторов!

И наконец, я хочу поблагодарить свою жену, Шарлотту, чья поддержка позволила мне посвятить время и силы написанию этой книги, а также Кристиана и Себастьяна, которые все время подбадривали меня.

---

1 *https://cppquiz.org*.

Предисловие

C++ – один из самых больших и старых языков программирования, которым активно пользуются. Он знаменит тем, что превратно истолковывает все поведения по умолчанию и, как в знаменитом в сообществе C++ примере, может заставить демонов вылетать у вас из носа[1]. Невозможно выбрать лучший язык для книги, посвященной головоломкам в программировании!

На примере 25 задач мы изучим, как C++ работает под капотом, и, в частности, уделим внимание нескольким его важным причудам. Чтобы извлечь максимум пользы из книги, вы должны иметь опыт работы на C++ и быть знакомы с основами языка, в том числе с простым объектно ориентированным программированием и шаблонами. Прочитав книгу, вы будете более глубоко разбираться в таких вопросах, как инициализация, время жизни, разрешение перегрузки, неявные преобразования, наследование, неопределенное и неспецифицированное поведение и др. Но, на мой взгляд, более важно то, что вы заинтересуетесь, как на самом деле работает C++, пусть даже в одной книге можно дать ответы лишь на немногие вопросы.

Как пользоваться этой книгой

Книга содержит 25 задач по C++ с ответами и объяснениями. Большая их часть представляет собой законченные программы, которые, согласно стандарту C++, дают вполне определенный результат. Но некоторые приводят к ошибкам компиляции и даже к неопределенному поведению. Ваша задача – понять, что произойдет после того, как вы откомпилируете и выполните каждую задачу, воспользовавшись отвечающей стандарту реализацией C++.

Возьмем для примера следующую гипотетическую задачу. Это полная программа на C++ с функцией `main`:

[1] *http://catb.org/jargon/html/N/nasal-demons.html*.

```
#include <iostream>
int main()
{
    std::cout << (1 < 2);
}
```

Ваша задача – прочитать код и попытаться угадать, что выведет откомпилированная программа. Обязательно подумайте сами, прежде чем переворачивать страницу в поисках ответа!

Хочу обратить внимание на несколько технических деталей.

- Для краткости я объявляю `main` без параметров (`argc`, `argv`) и не возвращаю явно значение. То и другое необязательно, а без них задачи немного проще читать.
- Во всех задачах я использую `struct`, а не `class`. Семантической разницы между ними нет – просто в `struct` по умолчанию подразумевается видимость членов `public`, а не `private`, поэтому нам не нужно вставлять всюду `public:`.
- Как всегда в C++, значения типа `bool` по умолчанию печатаются как `1` и `0`, а не `true` и `false`.

Эта программа печатает `1` (представляющую `true`), потому что `1` меньше `2`.

Но получить правильный ответ – только полдела. А вторая половина – понять, *почему* программа работает именно так, а не иначе. Задачи убеждают в том, что нужно лучше изучать, как C++ работает за кулисами. Я призываю вас вчитываться в объяснения, пока не достигнете полного понимания.

Неопределенное поведение

В некоторых задачах может иметь место неопределенное поведение. Так называется ситуация, когда при выполнении программы происходит что-то ужасное, чего компилятор не может (точнее говоря, не обязан) обнаружить. Например, мы можем обратиться к элементу за пределами массива, или арифметическое выражение, содержащее целые со знаком, может привести к переполнению. В таких случаях стандарт C++ не налагает никаких ограничений на реализацию, и случиться может все что угодно, включая демонов, вылетающих из носа. Если в задаче имеется неопределенное поведение, то вы должны не только идентифицировать его, но и предположить, что произойдет на практике в типичной системе. Действительно ли из носа

начнут вылетать демоны или случится что-то более приземленное и определенное?

Снова рассмотрим пример:

```
#include <iostream>
#include <limits>
int main()
{
    std::cout << std::numeric_limits<int>::max() + 1;
}
```

Переполнение в целочисленной арифметике со знаком – неопределенное поведение, поэтому, обнаружив его, вы на полпути к решению!

Строить догадки о том, что произойдет в случае неопределенного поведения, – занятие неблагодарное. Поэтому поступим иначе. В любой задаче с неопределенным поведением вторая половина – выяснить, что произойдет при запуске программы на своем компьютере. На моем компьютере целые со знаком представляются в дополнительном коде (как во всех реализациях, отвечающих стандарту C++20 и более поздним), и мой процессор не возбуждает исключения в случае переполнения. Поэтому, когда я прибавляю `1` к наибольшему положительному целому, происходит оборачивание и получается наименьшее отрицательное целое. Поскольку в моей системе тип `int` 32-разрядный, программа печатает `-2147483648`. Точное значение вам ни к чему, но если вы догадались, что будет напечатано наименьшее отрицательное целое, значит задача решена!

Не делайте этого дома

Угадывание или проверка того, что происходит в случае неопределенного поведения, – интересное упражнение, которое может пополнить ваши знания о работе C++ на своей платформе. Также это поможет распознавать определенные типы ошибок в реальных программах. Но не делайте никаких предположений о своих реальных программах на основе этих находок! Ваши предположения могут оказаться ложными на других компьютерах, после обновления компилятора или при компиляции с другими параметрами оптимизации. Компилятору даже разрешено удалять проверку ошибок из кода, если он может доказать, что имеет место неопределенное поведение!

Неспецифицированное и зависящее от реализации поведение

Стандарт C++ не все определяет строго, он оставляет некоторую свободу реализации. Вот несколько примеров:

- конкретные размеры целых типов;
- порядок вычисления аргументов функции;
- порядок инициализации глобальных переменных.

Это позволяет каждой реализации принимать решения, наиболее подходящие в конкретной системе.

В большинстве программ какое-то неспецифицированное или зависящее от реализации поведение присутствует, и это не ошибка. В отличие от неопределенного поведения демоны из носа не полетят. Просто разные реализации могут вести себя немного по-разному, не выходя за рамки допустимого поведения.

Если в задаче имеет место неспецифицированное или зависящее от реализации поведение, попытайтесь догадаться, как поведет себя программа в типичном случае.

Эксперименты с кодом

Самая важная часть изучения всего, связанного с программированием, – самостоятельные эксперименты. Код из этой книги можно найти на ее домашней странице на сайте The Pragmatic Bookshelf[1]. Вы можете собрать его локально, открыв файл `CMakeLists.txt` в своей любимой IDE или в командной строке:

```
mkdir build
cd build
cmake ..
cmake --build .
```

Проект содержит по одному `cpp`-файлу на задачу, и каждый из них транслируется в один двоичный файл, названный так же, как соответствующая задача в книге.

Можете также повозиться с кодом непосредственно в браузере, скопировав его в онлайновый компилятор. Я горячо рекомендую

1 *https://pragprog.com/titles/akbrain*.

сайт Compiler Explorer[1], где вы можете выбрать разные компиляторы и сравнить версии и архитектуры, попробовать различные параметры оптимизации, добавить другие флаги компилятора, параметры проверки кода и т. д.

Итак, приступим! Да – и берегитесь демонов.

---

[1] *https://godbolt.org*.

Задача 1

Сколько градусов?

how-many-degrees.cpp

```
#include <iostream>

struct Degrees
{
    Degrees() : degrees_(0)
    {
        std::cout << "Default constructed\n";
    }
    Degrees(double degrees) : degrees_(degrees)
    {
        std::cout << "Constructed with " << degrees_ << "\n";
    }
    double degrees_;
};
struct Position
{
    Position() : latitude_{1} { longitude_ = Degrees{2}; }
    Degrees latitude_;
    Degrees longitude_;
};

int main()
{
    Position position;
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

Программа печатает следующий результат:

```
Constructed with 1
Default constructed
Constructed with 2
```

Обсуждение

В структуре Position два члена типа Degrees. Один инициализируется в списке инициализации членов, второй в теле конструктора. Тогда почему мы видим *три* сконструированных объекта Degrees?

Все члены класс инициализируются до входа в тело конструктора. Если вы явно инициализируете член, как мы поступили с latitude_, то именно эта инициализация и будет использована. Если нет, как в случае с longitude_, то член будет инициализирован по умолчанию, поэтому мы видим, что напечатана строка Default constructed. После того как все члены инициализированы, мы входим в тело конструктора, инициализируем временный объект Degrees{2}, а затем присваиваем его копированием ранее инициализированному по умолчанию члену longitude_.

Заметим, что вместо явной инициализации члена в списке инициализации членов можно было бы использовать инициализатор по умолчанию прямо в объявлении члена:

examples/default-member-initializer/default-member-initializer.cpp

```
struct Position
{
    Position() { longitude = Degrees{2}; }
    Degrees latitude{1}; // инициализатор члена по умолчанию
    Degrees longitude;
};
```

Использование инициализатора члена по умолчанию может быть полезной идеей, если конструкторов несколько. Тогда не нужно помнить о включении этого члена в каждый список инициализации членов.

Для дополнительного чтения

Конструкторы и списки инициализации членов
https://en.cppreference.com/w/cpp/language/constructor.

Задача 2

Теория струн

string-theory.cpp

```
#include <iostream>
#include <string>

void serialize(const void*) { std::cout << "const void*"; }
void serialize(const std::string&) { std::cout << "const string&"; }
int main()
{
    serialize("hello world");
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

Программа печатает следующий результат:

```
const void*
```

Обсуждение

Почему передача строки функции `serialize` приводит к вызову перегруженного варианта, который принимает указатель на `void`, а не к вызову варианта, принимающего строку?

Когда мы вызываем функцию, имеющую несколько перегруженных вариантов, компилятор применяет процесс *разрешения перегрузки*, чтобы понять, какой вариант самый подходящий. Для этого компилятор пытается преобразовать каждый аргумент функции к типу соответствующего параметра для каждого перегруженного варианта. Одни преобразования считаются лучше других, а самое лучшее – когда аргумент уже имеет правильный тип.

Перегруженные варианты, в которых все аргументы можно успешно преобразовать, помещаются в множество *подходящих функций*. Затем компилятор должен определить, какой вариант выбрать из этого множества. Если некоторый перегруженный вариант имеет лучшее преобразование, чем другие, хотя бы для одного аргумента и не худшее для всех остальных, то этот вариант считается *лучшей подходящей функцией* и является результатом процесса разрешения перегрузки. Если ни один вариант не лучше всех остальных, то вызов считается недопустимым и не компилируется.

Рассмотрим пример:

```
serialize(int, int); // 1
serialize(float, int); // 2
```

При таких двух перегруженных вариантах предположим, что имеется вызов:

```
serialize(1, 2);
```

Оба перегруженных варианта `serialize` являются подходящими. Но для первого преобразование первого аргумента лучше (`int` → `int` лучше, чем `int` → `float`), а для второго – не хуже (`int` → `int` в обоих вариантах), поэтому он выбирается как лучшая подходящая функция.

Задача немного проще этого примера, потому что оба перегруженных варианта `serialize` имеют только один параметр. Первый принимает `const void *`, второй – `const std::string&`. Как выглядит преобразование для каждого варианта? `std::string` – класс из стандартной библиотеки. Обычно память для него выделяется из кучи (если только строка не очень мала), а строка может расти или модифицироваться каким-то другим способом во время выполнения.

Однако строка `"hello world"` не объект класса `std::string`, а простой строковый литерал. Строковые литералы – это массивы `char` в стиле C, которые включены в двоичный файл компоновщиком и не могут быть модифицированы во время выполнения. Строковый литерал имеет тип «массив из `n` элементов типа `const char`». Литерал `"hello world"` состоит из 11 символов плюс завершающий `\0`, т. е. имеет тип «массив из `12` элементов типа `const char`».

Поскольку аргумент `"hello world"` принадлежит не типу `const void*` и не типу `std::string`, а типу «массив из `12` элементов типа `const char`», то для обоих перегруженных вариантов необходимо преобразование. Если существует неявное преобразование типа фактического аргумента к типу формального параметра, то перегруженный вариант добавляется в множество подходящих функций. В противном случае вариант игнорируется.

Рассмотрим первый перегруженный вариант и выясним, можно ли неявно преобразовать «массив из `12` элементов типа `const char`» в `const void *`. Первым делом массив преобразуется в указатель. Любой ««массив из `N` элементов типа `T`» можно преобразовать в «указатель на `T`», указывающий на его первый элемент. Следовательно, наш «массив из `12` элементов типа `const char`» преобразуется в «указатель на `const char`».

Далее, любой «указатель на *cv* `T`» (где *cv* означает `const`, `volatile`, `const volatile` или отсутствие любого из этих модификаторов) можно преобразовать в «указатель на *cv* `void`». Стало быть, наш «указатель на `const char`» преобразуется в «указатель на `const void`», т. е. именно то, чего ожидает первый перегруженный вариант.

Заметим, что в этой последовательности преобразований не было ни конструкторов, ни функций преобразования. То есть это *последовательность стандартных преобразований*, а не *последовательность определенных пользователем преобразований*. Это окажется важно позже.

Теперь рассмотрим второй перегруженный вариант и выясним, можно ли неявно преобразовать «массив из `12` элементов типа `const`

char» в «ссылку на const std::string». В типе std::string имеется конструктор std::string(const char* s), которым мы можем воспользоваться. Сначала преобразуем «массив из 12 элементов типа const char» в «указатель на const char», как и раньше. Затем передадим его конструктору std::string и получим в ответ объект std::string, содержащий копию строкового литерала. Параметр типа const std::string& можно непосредственно связать с нашим аргументом типа std::string.

Заметим, что для этого нам пришлось использовать конструктор, т. е. мы имеем последовательность определенных пользователем преобразований, а не последовательность стандартных преобразований. Неважно, что std::string – тип из стандартной библиотеки, он все равно считается определенным пользователем. Правила одинаковы и для вас, и для стандартной библиотеки.

Итак, компилятор нашел допустимую последовательность преобразований нашего «массива из 12 элементов типа const char» в тип параметра каждого перегруженного варианта и должен решить, какая последовательность лучше:

Последовательность преобразований для void serialize(const void*)

const char[12] → const char * → const void * — Стандартное преобразование, новые объекты не создаются

Последовательность преобразований для serialize(const std::string&)

const char[12] → const char * → std::string — Определенное пользователем преобразование, создается объект типа std::string!

Для вызова варианта с const void* нам нужна только последовательность стандартных преобразований (верхняя). А для вызова варианта с std::string нужна последовательность определенных пользователем преобразований (нижняя), в которой встречается создание нового временного объекта std::string. Последовательность стандартных преобразований всегда *лучше* последовательности определенных пользователем преобразований, поэтому вызывается первый перегруженный вариант и печатается const void*.

Для дополнительного чтения

Разрешение перегрузки
https://en.cppreference.com/w/cpp/language/overload_resolution.

std::string
https://en.cppreference.com/w/cpp/string/basic_string.

Строковый литерал
https://en.cppreference.com/w/cpp/language/string_literal.

https://t.me/it_boooks/2

Задача 3

Хакнем планету

hack-the-planet.cpp

```
#include <iostream>

int getUserId() { return 1337; }
void restrictedTask1()
{
    int id = getUserId();
    if (id == 1337) { std::cout << "did task 1\n"; }
}
void restrictedTask2()
{
    int id;
    if (id == 1337) { std::cout << "did task 2\n"; }
}
int main() {
    restrictedTask1();
    restrictedTask2();
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

Поведение программы не определено! Но она может напечатать такой результат:

```
did task 1
did task 1
```

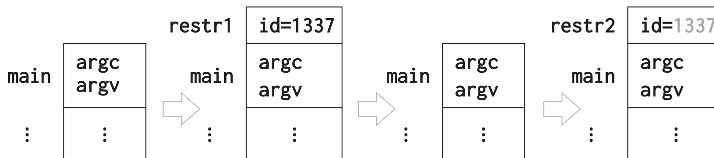
Обсуждение

Переменная `id` в функции `restrictedTask2` не была инициализирована, ее значение неизвестно. Использование ее значения и есть неопределенное поведение. Когда поведение программы не определено, может случиться все что угодно; стандарт C++ не дает никаких гарантий. Даже та часть программы, которая *предшествует* моменту, когда мы прочитали значение `id`, не определена!

Однако если вы запустите эту программу на своем компьютере, то она, скорее всего, напечатает оба сообщения, `did task 1` и `did task 2`, по крайней мере, если была откомпилирована без оптимизации. То есть значение `1337` волшебным образом телепортировалось из `restrictedTask1` в `restrictedTask2`! Как такое могло произойти?

В большинстве систем для размещения локальных переменных используется стек. В функции `restrictedTask1` имеется локальная переменная `id`, для которой она выделяет место в своем кадре стека. Так получилось, что в функции `restrictedTask2` столько же локальных переменных и точно таких же типов (одна типа `int`), поэтому структура ее кадра стека в точности такая же, как у `restrictedTask1`.

Пока мы находились в `main`, стек вырос до определенной точки – см. самую левую часть рисунка ниже. После этого мы вызываем `restrictedTask1`, стек растет, `restrictedTask1` резервирует в своем кадре стека место для `id` и инициализирует ее значением `1337`. Затем управление возвращается `main`, и стек снова сжимается.



Далее мы вызываем `restrictedTask2`, стек снова растет, и `restrictedTask2` резервирует в своем кадре стека место для `id`, но не инициализирует ее. Однако кадр стека `restrictedTask2` оказывается точно там же, где раньше был кадр стека `restrictedTask1`. Между вызовами функций стек не очищается, поэтому содержимое кадра

стека `restrictedTask2` такое, в каком его оставила `restrictedTask1`, и, в частности, в позиции локальной переменной `id` находится значение `1337`. Стало быть, `id` содержит `1337` в обеих функциях.

Но если включить оптимизацию, то `restrictedTask1` и `restrictedTask2`, вероятно, будут встроены в `main`, и для их вызова стек не будет использоваться вовсе. Можете попробовать сами, откомпилировав код с флагом `-O2` (GCC/Clang) или `/O2` (MSVC). На моей машине x86_64 с ОС Linux при компиляции GCC с флагом `-O2` печатается `did task 1`, а при компиляции Clang программа аварийно завершается с ошибкой сегментации. А как на вашей?

Избегайте неинициализированных переменных

Чтобы избежать вылетающих из носа демонов, уязвимостей типа показанной выше и вообще мусорных данных, всегда инициализируйте переменные перед использованием. Об этом легко забыть, но помощь придет! Прежде всего при компиляции всегда включайте предупреждения. В этом конкретном случае об упущении предупредят GCC, Clang и MSVC, равно как и инструменты типа clang-tidy. Но это только потому, что нетрудно доказать, что `id` не была инициализирована к моменту использования. А зачастую мы об этом не знаем до этапа выполнения. И вот тут на помощь приходят контролеры (sanitizer).

Разные контролеры служат разным целям, но все тем или иным способом наблюдают за вашим кодом во время выполнения, стремясь обнаружить проблемы, которые невозможно обнаружить на этапе компиляции. Одним таким контролером является MemorySanitizer, призванный отлавливать использование неинициализированной памяти. Если запустить программу с включенным MemorySanitizer (передайте компилятору Clang параметр `-fsanitize=memory`), то она напечатает нечто подобное:

```
==1416660==WARNING: MemorySanitizer: use-of-uninitialized-value
#0 0x5603d43af5eb in restrictedTask2() example.cpp:14:9
#1 0x5603d43af64d in main example.cpp:19:5
```

Контролер сообщает, что используется неинициализированная память, и печатает трассу стека, показывающую, где это произошло.

Контролеры могут сильно замедлять работу программы, поэтому обычно для них заводится отдельная конфигурация сборки. Контролер может обнаружить проблему, только если она действительно возникала в процессе выполнения, поэтому постарайтесь прогонять в этом режиме как можно больше своих тестов.

Рекомендации

- Всегда включайте предупреждения. Флаги `-Wall -Wextra -Wpedantic` – неплохая отправная точка для GCC/Clang, а флаг `/W4` – для MSVC.
- Если ваша IDE поддерживает интеграцию с инструментами линтинга, например clang-tidy, включите ее. Иногда такие инструменты способны сообщить о проблемах еще до компиляции кода.
- Включайте режим трактовки предупреждений как ошибок, по крайней мере в конвейере непрерывной интеграции, чтобы не пропустить никаких предупреждений.
- Включайте в сборки с контролерами как можно больше контролеров и собирайте свои тесты именно так.

Для дополнительного чтения

Неопределенное поведение
https://en.cppreference.com/w/cpp/language/ub.

Olve Maudal, Jon Jagger «Deep C (and C++)»
https://www.slideshare.net/olvemaudal/deep-c.

Список контролеров
https://github.com/google/sanitizers.

Задача 4

На пути к глобализации

going-global.cpp

```
#include <iostream>

int id;
int main()
{
    std::cout << id;
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

Программа напечатает следующий результат:

```
0
```

Обсуждение

Эта задача похожа на задачу 3; в обеих используется значение неинициализированной переменной `id`. Но, в отличие от задачи 3, здесь нет неопределенного поведения, и программа печатает 0. Почему?

Разница в том, что в задаче 3 переменные `id` локальны, т. е. возникают из небытия и возвращаются туда всякий раз, как функция вызывается. Говорят, что у них *автоматическая продолжительность хранения*. С другой стороны, в этой задаче `id` – глобальная переменная. Точнее, `id` – нелокальная переменная со *статической продолжительностью хранения*, т. е. ее время жизни совпадает с временем жизни всей программы.

В отличие от автоматической продолжительности хранения переменные со статической продолжительностью хранения инициализируются на этапе инициализации программы. В простейшем случае переменной типа `int` без инициализатора имеет место *инициализация нулем*. Если бы у `id` был инициализатор, например `int id = 2;`, то мы бы сказали, что она *инициализирована константой* `2`. (Для инициализации нулем и константой имеется общий термин – *статическая инициализация*.)

Почему переменные со статической продолжительностью хранения без инициализатора инициализируются нулем, а переменные с автоматической продолжительностью хранения – нет? В общем случае инициализация переменной обходится не даром. Для этого необходимо выполнить некий код. Например, если в функции имеется определение локальной переменной `int id = 0;`, то в большинстве систем будет сгенерирован код, который присваивает ей значение 0 при каждом вызове функции. C++ не может гарантировать, что стоимость этой операции нулевая или пренебрежимо малая, а одна из проектных целей C++ – не платить за то, что не используется. C++ не желает оставлять место для появления более производительного языка. Поэтому если вы решите опустить инициализатор и просто определите `int id;`, то C++ не заставит вас заплатить эту цену.

Другое различие между локальными и нелокальными переменными – область видимости. Локальная переменная доступна только в том блоке, в котором определена. Напротив, нелокальная переменная доступна из любого места файла, а если она объявлена в заголовочном файле, то из любого места программы, в которое этот файл включен. Так что, опустив инициализатор для локальной переменной, вы сохраняете точный контроль над тем, где она может использоваться. Быть может, вы оставили ее неинициализированной, потому что собираетесь инициализировать ее парой строк ниже. Но если вы опустите инициализатор для нелокальной переменной, то рассуждать о том, где она будет использована, гораздо сложнее.

Сравним нелокальные переменные со статической продолжительностью хранения и локальные переменные с автоматической продолжительностью хранения:

	Стоимость инициализации	**Область видимости**
Нелокальная статическая	Однократно	Файл или программа целиком
Локальная автоматическая	При каждом входе в блок	Блок, в котором определена

Контролировать использование нелокальных переменных со статической продолжительностью хранения труднее, а их инициализация обходится дешево, поэтому C++ производит ее автоматически. Но насколько дешево? Рассмотрим, к примеру, архитектуру x86. Выше я сказал, что мы платим за инициализацию однократно, но на самом деле не платим ничего! При запуске программа выделяет память для таких вещей, как стек, куча, программный код и, что для нас самое важное, для глобальных данных. В Linux и Windows страница физической памяти, возвращенная операционной системой, заполняется нулями, чтобы предотвратить чтение данных, оставленных программой, которой эта страница принадлежала ранее. Компилятор и компоновщик могут сгруппировать все неинициализированные глобальные переменные в одну секцию, которая называется `.bss`. Когда программа запускается и выделяет память для секции `.bss`, она уже будет заполнена нулями. То есть эти переменные инициализируются нулями даром!

Примечание

У слова `static` в C++ много значений. Наличие или отсутствие у переменной `id` спецификатора `static` не влияет на ее статическую продолжительность хранения. Написав `static int id;` вместо `int id;` для глобальной переменной, мы всего лишь получим внутреннее связывание. Это означает, что с ней нельзя установить связь из других единиц трансляции (других `cpp`-файлов). Но в любом случае она имеет статическую продолжительность хранения.

Для дополнительного чтения

Инициализация нулями

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

Еще об инициализации глобальных переменных: от программы к процессу

https://www.youtube.com/watch?v=fGnbGX88z3Y.

https://t.me/it_boooks/2

Задача 5

Деструктивные отношения

a-destructive-relationship.cpp

```
#include <iostream>
#include <memory>

struct Widget
{
    virtual void draw() { std::cout << "Widget draw\n"; }
    virtual ~Widget() { std::cout << "Widget destructor\n"; }
};
struct Button : public Widget
{
    void draw() override { std::cout << "Button draw\n"; }
    ~Button() override { std::cout << "Button destructor\n"; }
};
int main()
{
    std::unique_ptr<Widget> widget = std::make_unique<Button>();
    widget->draw();
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

Программа напечатает следующий результат:

```
Button draw
Button destructor
Widget destructor
```

Обсуждение

При вызове `draw` программа печатает только `Button draw`. Но когда `unique_ptr` выходит из области видимости и вызывается деструктор, программа печатает *оба* сообщения: `Button destructor` и `Widget destructor`. Почему?

Обычная техника реализации динамического полиморфизма – использование виртуальных функций. Мы объявляем класс, содержащий одну или несколько функций с ключевым словом `virtual`, которое позволяет другим классам переопределять эти функции. Если затем вызвать виртуальную функцию через указатель на базовый класс, который на самом деле указывает на объект производного класса, то будет использован динамический тип объекта и вызвана переопределенная функция.

В нашем примере предположим, что имеется каркас UI, включающий класс `Widget`, который определяет общее поведение, и классы конкретных элементов типа `Button`, `Text` и `Image`, которые переопределяют поведение рисования в виртуальной функции `draw`.

Когда `draw` вызывается через указатель на базовый класс `Widget`, который в действительности указывает на производный класс `Button`, управление получает не `Widget::draw`, а `Button::draw`, печатающая сообщение `Button draw`. Однако поведение деструктора иное; как мы видим, сначала печатается `Button destructor`, а затем *еще и* `Widget destructor`. Что происходит?

При уничтожении объекта должны быть уничтожены все его части. В данном случае `Button` наследует `Widget`, поэтому каждый объект `Button` состоит из объекта `Widget` и некоторых членов `Button` (в данном случае таковых нет). Говорят, что `Button` содержит *подобъект* `Widget`. Если бы `Button` содержал член типа `Widget`, а не наследовал этому типу, то он также имел бы подобъект типа `Widget`. В любом случае эти подобъекты должны быть уничтожены, когда уничтожается содержащий

их объект, иначе возможна утечка ресурсов. По счастью, C++ заботится об этом сам; нам не нужно вручную вызывать деструктор `Widget` из деструктора `Button`.

Отметим также, что сначала выполняется тело деструктора, а только потом вызываются деструкторы подобъектов. Тем самым гарантируется, что в деструкторе класса мы по-прежнему можем использовать члены базовых классов. Поэтому программа сначала печатает `Button destructor`, а потом `Widget destructor`.

Дополнительный вопрос

Что произошло бы, если бы в классе `Widget` деструктор не был объявлен как `virtual`, и в каких случаях это стало бы проблемой?

Согласно стандарту C++, если статический тип удаляемого объекта отличен от его динамического типа, то статический тип обязан быть базовым классом динамического типа и должен иметь виртуальный деструктор. В противном случае поведение не определено.

Таким образом, если `Widget` не объявляет свой деструктор виртуальным и мы удаляем объект `Button` через указатель на `Widget`, то имеет место неопределенное поведение. На практике мы обычно встречаем, что вместо деструктора `Button` вызывается деструктор `Widget`, что приводит к следующим проблемам:

- никакие члены производного класса не уничтожаются, что потенциально может стать причиной утечек памяти и других ресурсов;
- если в производном классе имеется определенный пользователем деструктор, то корректность программы, как правило, зависит от логики, выполняемой в нем в момент уничтожения объекта.

Но, как бы то ни было, даже если эти проблемы не касаются вашей конкретной программы и во время тестирования кажется, что все хорошо, вы все равно должны объявлять деструктор виртуальным, чтобы избежать неопределенного поведения.

Рекомендация

Благодаря автоматическому уничтожению подобъектов в C++ получила распространение техника управления ресурсами на основе идиомы «захват ресурса является инициализацией» (Resource Acquisition Is Initialization – RAII), когда ресурсы объявляются как члены типа (например, `unique_ptr`), деструктор которого отвечает за их освобождение. Эта техника работает и в случае, когда вы наследуете такому классу.

Для дополнительного чтения

CppReference о деструкторах
https://en.cppreference.com/w/cpp/language/destructor.

Стандарт C++ о деструкторах и подобъектах
https://timsong-cpp.github.io/cppwp/std20/class.dtor#14.

Задача 6

Кто первый?

whos-up-first.cpp

```cpp
#include <iostream>

struct Resource
{
    Resource()
    {
        std::cout << "Resource constructed\n";
    }
};
struct Consumer
{
    Consumer(const Resource &resource)
    {
        std::cout << "Consumer constructed\n";
    }
};
struct Job
{
    Job() : resource_{}, consumer_{resource_} {}
    Consumer consumer_;
    Resource resource_;
};
int main() {
    Job job;
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

Программа напечатает следующий результат:

```
Consumer constructed
Resource constructed
```

Обсуждение

В классе `Job` два члена, `consumer_` и `resource_`, объявленные именно в таком порядке. Однако в списке инициализации членов в конструкторе (часть `: resource_{}, consumer_{resource_}`) они инициализируются в противоположном порядке. В каком же порядке на самом деле производится инициализация?

C++ всегда инициализирует члены в том порядке, в каком они объявлены в классе, а не в том, в каком перечислены в списке инициализации членов в конструкторе. Тем самым гарантируется, что инициализация всегда производится в детерминированном порядке, даже если в разных конструкторах члены перечислены в разном порядке. Поэтому, хотя и кажется, что `resource_` инициализируется в конструкторе первым, на самом деле первым инициализируется `consumer_`.

Тот факт, что члены инициализируются детерминированно, важен, когда один член зависит от другого. Например, в нашей задаче потребитель (consumer) зависит от ресурса (resource) в конструкторе. И все вроде бы хорошо, но, поскольку в действительности `consumer_` инициализируется раньше `resource_`, конструктору `Consumer` передается еще не сконструированный объект `Resource`. Если бы мы попытались хоть как-то использовать его, то получили бы неопределенное поведение.

Обратное также верно. Члены уничтожаются в порядке, противоположном их объявлению в классе. Порядок уничтожения может быть так же важен, как порядок конструирования, потому что один член может зависеть от другого в деструкторе или же могут существовать другие зависимости от времени жизни. Предположим, например, что `Consumer` в этой задаче выполняет поток, в котором используется `Resource` и что этот `Consumer` ждет завершения потока в своем деструкторе. В текущей версии кода `resource_` был бы уничтожен, когда поток еще работает, и возник бы риск использования уже уничтоженного `resource_`, что привело бы к неопределенному поведению. Но если бы `consumer_` был объявлен *после* `resource_`, то поток в `consumer_` остановился бы раньше, чем был бы уничтожен `resource_`.

Перечисление членов не в том порядке в списке инициализации членов может запутать читателя, поэтому я рекомендую всегда использовать в списке инициализации членов тот же порядок, что в объявлении класса. GCC, Clang и MSVC выдают предупреждения (соответственно `-Wreorder`, `-Wreorder-ctor` и `C5038`), если порядок не согласован. Эти предупреждения разрешены, когда компилятор вызван с флагом `-Wall` (или `/Wall`), который я рекомендую задавать всегда. Разные компиляторы часто выдают различные предупреждения, поэтому имеет смысл попробовать несколько. Это может также помочь избежать случайного использования нестандартных платформенно зависимых вариаций C++, что затруднило бы последующий перенос проекта на другие платформы.

Рекомендации

- Всегда включайте предупреждения.
- Старайтесь компилировать разными компиляторами

Для дополнительного чтения

Конструкторы и списки инициализации членов
https://en.cppreference.com/w/cpp/language/constructor.

Задача 7

Все хорошее должно когда-то заканчиваться

all-good-things-must-come-to-an-end.cpp

```
#include <iostream>
#include <string>

struct Connection
{
    Connection(const std::string &name) : name_(name)
    {
        std::cout << "Created " << name_ << "\n";
    }
    ~Connection() {
        std::cout << "Destroyed " << name_ << "\n";
    }
    std::string name_;
};
Connection global{"global"};
Connection &get()
{
    static Connection localStatic{"local static"};
    return localStatic;
}
int main()
{
    Connection local{"local"};
    Connection &tmp1 = get();
    Connection &tmp2 = get();
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

Программа напечатает следующий результат:

```
Created global
Created local
Created local static
Destroyed local
Destroyed local static
Destroyed global
```

Обсуждение

В задаче 3 и в задаче 4 мы узнали о различиях во времени жизни и инициализации между глобальной и локальной переменной `id`:

```
int id; // глобальная переменная, один экземпляр на всю программу

void f() {
   int id = 2; // локальная переменная, создается заново при каждом вызове `f`
}
```

В этой задаче добавлено два новых интересных аспекта времени жизни объекта:

- мы больше не инициализируем переменные простой константой; имеется конструктор с побочным эффектом;
- мы ввели локальную переменную `localStatic` со статической продолжительностью хранения.

Как же изменилось поведение после добавления конструктора и локальной статической переменной? Сначала рассмотрим глобальную переменную `global`. Как и глобальная переменная, которую мы видели раньше, она имеет статическую продолжительность хранения, т. е. один-единственный экземпляр остается в живых с момента инициализации и до момента уничтожения в самом конце программы.

Ранее мы видели, что глобальная переменная `int id;` без инициализатора инициализировалась нулями. Мы также отметили, что глобальная переменная с инициализатором `int id = 2;` была бы инициализирована константой. То и другое – примеры статической инициализации, которая производится на этапе запуска программы. С другой стороны, глобальная переменная `global` инициализируется неконстантным выражением. Конструктор `Connection` не является `constexpr` (и не может быть таковым, потому

что печатает на `std::cout`). Объекты, которые нельзя инициализировать статически, инициализируются динамически, и это происходит позже.

Когда именно производится динамическая инициализация? Принято считать, что динамическая инициализация производится до вызова `main`, – и тому есть веская причина: это простой и распространенный способ реализовать динамическую инициализацию. Например, в Linux имеется функция `__libc_start_main`, которая получает управление до `main` и вызывает глобальные конструкторы, прежде чем вызвать вашу функцию `main`.

Стандарт C++ не требует, чтобы динамическая инициализация производилась до `main`, а просто разрешает это. Однако стандарт требует, чтобы динамическая инициализация глобальных переменных была произведена до вызова любых функций, определенных в той же единице трансляции (translation unit – TU). (Единица трансляции – это просто одиночный `cpp`-файл после включения всех заголовочных файлов директивами `#include`.) Таким образом, в этой задаче, где `main` определена в той же TU, что и `global`, гарантируется, что `global` будет инициализирована раньше `main`!

Полные правила порядка инициализации длинные и сложные. На сайте CppReference.com имеется введение в эту тему[1], а все детали изложены в стандарте C++ в разделе [basic.start][2].

Теперь обратимся к переменной `localStatic`. Как и `global`, она имеет статическую продолжительность хранения, поэтому существует всего один экземпляр `localStatic` со временем жизни от момента инициализации и до конца программы. В отличие от глобальных, локальные статические переменные инициализируются не в момент запуска программs, а при первом прохождении потока управления через их объявления. Поэтому, хотя продолжительность хранения `localStatic` статическая, она остается неинициализированной до первого вызова `get`.

Наконец, `local` – простая локальная переменная с автоматической продолжительностью хранения; она инициализируется при каждом прохождении потока управления через ее объявление.

Теперь посмотрим, как выполняется эта программа:

- программа запускается, и прямо перед вызовом `main` инициализируется переменная `global`;
- мы входим в `main`, и в этот момент инициализируется `local`;

---

[1] *https://en.cppreference.com/w/cpp/language/initialization*.

[2] *https://timsong-cpp.github.io/cppwp/std20/basic.start*.

- мы в первый раз вызываем `get`, и в этот момент инициализируется `localStatic`;
- мы вызываем `get` во второй раз, но, поскольку `localStatic` уже инициализирована, она не инициализируется повторно.

Но это еще не все! Нужно понять, когда наши три объекта уничтожаются. По счастью, правила уничтожения гораздо проще. Локальные объекты с автоматической продолжительностью хранения уничтожаются в конце той области видимости, в которой объявлены. Объекты со статической продолжительностью хранения уничтожаются в конце программы в порядке, противоположном порядку их создания:

- когда мы покидаем `main`, `local` выходит из области видимости и уничтожается;
- `localStatic` была последним инициализированным объектом со статической продолжительностью хранения, поэтому уничтожается она первой;
- `global` была инициализирована первой и уничтожается последней.

Использование статических локальных объектов вместо глобальных – хороший способ точнее контролировать время жизни «глобальной» переменной. Вместо того чтобы ссылаться на глобальный объект всякий раз, как в нем возникает необходимость, мы вызываем функцию, чтобы получить ссылку на него, и таким образом точно определяем момент инициализации объекта! Особенно это полезно, когда имеется несколько глобальных объектов, зависящих друг от друга.

И напоследок предостережение относительно глобальных объектов, определенных в нескольких TU. Инициализация глобальных объектов в пределах *одного* TU производится в определенном порядке, но глобальные объекты в *разных* TU могут быть инициализированы в любом порядке. Например, вы могли бы ожидать, что следующая программа напечатает `1`, но на моем компьютере она печатает `0`.

examples/siof/Value.h

```
struct Value
{
    Value(int i_) : i(i_) {}
    Value(const Value &other) : i(other.i) {}
    int i;
};
```

examples/siof/source1.cpp

```
#include "Value.h"
#include <iostream>

extern Value value1;
Value value2{value1}; // Нет гарантии, что `value1` была инициализирована!
int main()
{
	std::cout << value2.i;
}
```

examples/siof/source2.cpp

```
#include "Value.h"

Value value1{1};
```

Эта проблема неопределенного порядка инициализации глобальных объектов в разных единицах трансляции часто называется «фиаско порядка статической инициализации» (Static Initialization Order Fiasco – SIOF)[1].

Рекомендации

- Избегайте глобальных переменных, зависящих друг от друга, особенно если они определены в разных файлах.
- Используйте clang-tidy, в ней имеется проверка `cppcoreguidelines-interfaces-global-init`, которая способна обнаружить некоторые случаи использования зависимых глобальных переменных.
- Вместо глобальных переменных используйте функции с локальными статическими переменными; это позволяет точнее контролировать время жизни.

[1] https://en.cppreference.com/w/cpp/language/siof.

Для дополнительного чтения

Еще о динамической инициализации: как запустить программу
https://www.youtube.com/watch?v=OGPmZzhDPYw.

CppReference.com об инициализации
https://en.cppreference.com/w/cpp/language/initialization.

Стандарт C++ об инициализации и завершении
https://timsong-cpp.github.io/cppwp/std20/basic.start.

Фиаско порядка статической инициализации (SIOF)
https://en.cppreference.com/w/cpp/language/siof.

Задача 8

Переместится или нет?

will-it-move.cpp

```
#include <iostream>

struct Member
{
};
struct WillItMove
{
    WillItMove() = default;
    WillItMove(WillItMove &&) = default;
    const Member constMember_{};
};
int main()
{
    WillItMove objectWithConstMember;
    WillItMove moved{std::move(objectWithConstMember)};
    std::cout << "It moved!\n";
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

Программа напечатает следующий результат:

```
It moved!
```

Обсуждение

`WillItMove` имеет константный член. Произвести перемещение из константного объекта невозможно, так почему же нам удалось переместить из `objectWithConstMember`?

В этой задаче мы сначала инициализируем переменную `WillItMove objectWithConstMember;`. Затем в следующей строке инициализируется переменная `WillItMove moved{std::move(objectWithConstMember)};`. В структуре `WillItMove` нет копирующего конструктора, поэтому для инициализации `moved` остается только одна альтернатива – перемещающий конструктор. Но как перемещающий конструктор может выполнить перемещение из `const Member constMember`?

Чтобы разобраться, пойдем снаружи внутрь.

Прежде всего что делает функция `std::move(objectWithConstMember)`? На самом деле практически ничего, она только преобразовывает l-значение `objectWithConstMember` в r-значение. Затем это r-значение используется для инициализации `moved`.

l-значения и r-значения

l-значения и r-значения – это способ классификации выражений. l-значением называется выражение, результатом вычисления которого является настоящий объект, обладающий идентичностью, перемещение из которого невозможно. Например, результатом вычисления выражения `objectWithConstMember` является объект `objectWithConstMember`. У него есть имя и место в памяти, и другие объекты еще могут его использовать, поэтому система типов не позволит что-то переместить из него. По традиции l-значения обычно встречаются в левой части оператора присваивания, отсюда и название – left-value (значение слева).

С другой стороны, r-значение – это выражение, результатом вычисления которого является нечто допускающее перемещение содержимого, например временный объект, до которого все равно никто не может добраться.

r-значения обычно встречаются в правой части оператора присваивания, отсюда и название – right-value (значение справа).

`std::move` преобразует l-значение в r-значение, т. е. говорит системе типов «все нормально, можешь перемещать из этого l-значения, мне оно больше не понадобится». С появлением `std::move` l-значения стали чаще встречаться *справа* от знака присваивания (внутри `std::move`), так что противопоставление l-значений и r-значений уже не так полезно, как раньше.

Смотрите статью «Lvalues, Rvalues, Glvalues, Prvalues, Xvalues, Help!»[1], в которой имеется развернутый обзор l-значений, r-значений и других категорий значений.

Заметим, что, хотя мы инициализировали переменную `moved` r-значением, это не значит, что для ее инициализации обязательно использовать перемещающий конструктор! Подойдет любой конструктор, способный привязаться к r-значению. Константную ссылку на l-значение можно связать с r-значением, поэтому можно было бы использовать и копирующий конструктор, если бы он у нас был. Но поскольку его нет, то перемещающий конструктор `WillItMove` – единственный вариант.

Перемещающий конструктор `WillItMove` явно определен как `default` в строке `WillItMove (WillItMove &&) = default;`, поэтому поговорим о том, что делает такой конструктор. Как он может переместить значение из константного члена?

Явный перемещающий конструктор по умолчанию непосредственно инициализирует все подобъекты соответствующими членами исходного объекта. По существу, он определяется примерно так:

examples/default-move/default-move.cpp

```
Class(Class &&other) :
    BaseClass1{std::move(other)},
    BaseClass2{std::move(other)},
    /* ... */
    member1_{std::move(other).member1_},
    member2_{std::move(other).member2_}
    /* ... */
    {}
```

[1] *https://blog.knatten.org/2018/03/09/lvalues-rvalues-glvalues-prvalues-xvalues-help/*.

Так что для нашего класса WillItMove он имеет вид:

examples/default-move/default-move.cpp

```
WillItMove(WillItMove && other) :
    constMember_{std::move(other).constMember_}{}
```

Повторим еще раз, что, даже если std::move(other).constMember_ является r-значением, мы не обязаны использовать перемещающий конструктор Member для инициализации constMember_. Подойдет как копирующий, так и перемещающий конструктор при условии, что он может привязаться к std::move(other).constMember_. Поскольку мы не объявили никаких специальных функций-членов в классе Member, он неявно получает копирующий и перемещающий конструкторы по умолчанию. Может ли хотя бы один из них привязаться к константному r-значению other.constMember_?

Неявный перемещающий конструктор по умолчанию для Member объявляется следующим образом:

```
Member(Member&&)
```

r-значение в форме ссылке на неконстанту не может быть связано с константным r-значением, поэтому мы не можем использовать перемещающий конструктор для непосредственной инициализации constMember_.

Неявный копирующий конструктор по умолчанию для Member объявляется следующим образом:

```
Member(const Member&)
```

l-значение в форме ссылки на константу может быть связано с константным r-значением, поэтому мы можем использовать копирующий конструктор для непосредственной инициализации constMember_.

Таким образом, явный перемещающий конструктор по умолчанию для WillItMove использует *копирующий*, а не перемещающий конструктор Member для инициализации constMember_.

Для дополнительного чтения

Перемещающие конструкторы
https://en.cppreference.com/w/cpp/language/move_constructor.

Копирующие конструкторы
https://en.cppreference.com/w/cpp/language/copy_constructor.

Непосредственная инициализация
https://en.cppreference.com/w/cpp/language/direct_initialization.

std::move
https://en.cppreference.com/w/cpp/utility/move.

Различие между неперемещающим конструктором и удаленным перемещающим конструктором
https://blog.knatten.org/2021/10/15/the-difference-between-no-move-constructor-and-a-deletedmove-constructor/.

Задача 9

Подсчет копий

counting-copies.cpp

```
#include <iostream>

struct Resource
{
    Resource() = default;
    Resource(const Resource &other)
    {
        std::cout << "copy\n";
    }
};
Resource getResource()
{
    return Resource{};
}
int main()
{
    Resource resource1 = getResource();
    Resource resource2{resource1};
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

Программа напечатает следующий результат:

```
copy
```

Обсуждение

Может показаться, что в этой программе производится три операции копирования. Одна – чтобы инициализировать объект, возвращаемый `getResource`, временным объектом `Resource{}`, другая – чтобы инициализировать `resource1` этим возвращенным объектом, и третья – чтобы инициализировать `resource2`. Однако при возврате из `getResource()` и инициализации `resource1` не было никакого копирования. Как такое возможно?

Сначала взглянем на предложение `return`, которое, по видимости, инициализирует возвращаемый объект временным объектом `Resource{}`. Выражение `Resource{}` в предложении `return` – это разновидность r-значения, называемая pr-значением, т. е. «чистым» (pure) r-значением. В задаче 8 мы впервые встретились с l-значениями и r-значениями и узнали, что r-значения – это выражения, из которых можно произвести перемещение, с которыми можно связать ссылки на r-значения и т. д. На самом деле есть два типа r-значений – x-значения и pr-значения.

- x-значение («eXpiring lvalue» – исчезающее l-значение) – это то, что мы получаем, когда вызываем `std::move` для l-значения. Например, в задаче 8 выражение `objectWithConstMember` является l-значением. Результатом его вычисления является настоящий объект, который где-то существует, имеет адрес, который можно получить, и т. д. После выполнения `std::move(objectWithConstMember)` мы получаем x-значение. Результатом его вычисления по-прежнему является настоящий объект, но разрешается также обращаться с ним как с r-значением и производить из него перемещение.
- С другой стороны, pr-значение – это «чистое» r-значение. Это выражение, которое инициализирует новый объект, а не просто преобразует существующий объект в r-значение.

В C++11 pr-значениями были неименованные временные объекты, и компилятору разрешалось пропускать их копирование. Форма пропуска копирования, при которой неименованный временный объект являлся операндом предложения `return`, как в этой задаче, называлась *оптимизацией неименованного возвращаемого значения*

(unnamed return value optimization – URVO) – неименованного, потому что применялась, когда функция возвращала неименованный временный объект. В C++17 отказ от копирования в таких ситуациях стал обязательным. Но, вместо того чтобы делать URVO обязательным, язык был изменен таким образом, что теперь даже и объекта, копирование которого можно было бы устранить, не существует! pr-значения больше не являются неименованными временными объектам, а просто представляют *идею объекта*, который будет материализован позже.

Еще одна форма оптимизации возвращаемого значения, до сих пор не являющаяся обязательной, называется *оптимизацией именованного возвращаемого значения* (named return value optimization – NRVO). Она применима, когда возвращается локальная переменная, а не pr-значение; в этом случае реализации *разрешается* опускать копирование, но это необязательно. NRVO применима, даже если копирующий конструктор имеет побочные эффекты; в этом случае побочные эффекты попросту не проявляются. Это тот редкий случай, когда оптимизация может изменить наблюдаемое поведение программы!

В примере ниже демонстрируется оптимизация неименованного и именованного возвращаемого значения:

examples/rvo/rvo.cpp

```
Resource getResource() {
    return Resource{}; // URVO (до C++17)
}
```

examples/rvo/rvo.cpp

```
Resource getResource()
{
    Resource resource;
    return resource; // NRVO
}
```

Теперь взглянем на выражение `Resource resource1 = getResource()`, которое, по видимости, копирует возвращенный объект в `resource1`. Вызов функции – это pr-значение, поэтому к нему применима та же идея. Вызов `getResource()` не приводит к созданию временного объекта, он просто представляет *идею* возвращаемого значения, а никакого объекта `Resource` не создается, пока `resource1` не будет им инициализирована.

Определение таких вещей, как «идея объекта», – это, конечно, прекрасно, но как осуществить это на практике? Ведь предполагается, что объект `Resource` окажется в стеке функции `main`, но фактически создающий его код находится в `getResource()`, которая не имеет доступа к кадру стека `main`.

Фокус, который большинство систем применяют, чтобы избежать копирования, заключается в том, что `main` резервирует место для `resource1` в своем кадре стека точно так же, как для любой другой локальной переменной. Затем `main` передает указатель на эту память в виде «секретного» дополнительного аргумента `getResource()`, как если бы та имела сигнатуру `getResource(Resource* ptr)`. Теперь `getResource()` может сконструировать объект непосредственно в этой области памяти.

Чтобы этот прием работал, вызывающая и вызываемая сторона должны следовать этому соглашению, а значит, оно должно быть частью системного двоичного интерфейса приложений (Application Binary Interface – ABI). Один из популярных ABI – Itanium C++ ABI[1], используемый, например, в Linux. И GCC, и Clang поддерживают этот ABI, поэтому функцию, откомпилированную одним, можно без опаски вызывать из функции, откомпилированной другим. В Itanium C++ ABI сказано:

> Если возвращаемое значение имеет тип класса, не тривиальный с точки зрения вызовов, то вызывающая сторона передает адрес неявного параметра. Вызываемая сторона конструирует возвращаемое значение в области по этому адресу.

Если попробовать это на сайте Compiler Explorer, выбрав архитектуру x86_64, то обнаружится, что текущие версии GCC и Clang генерируют в `main` такой код:

```
lea     rax, [rbp-4]
mov     rdi, rax
call    getResource()
```

`[rbp-4]` – это адрес `resource1` в кадре стека `main`. Этот адрес загружается в регистр `rdi`, который служит для передачи первого целого или указательного аргумента функции, после чего вызывается `getResource()`.

Понять, каким ABI руководствуется MSVC, труднее. Но Compiler Explorer показывает похожий код для x86_64:

1 *https://itanium-cxx-abi.github.io/cxx-abi/abi.html*.

```
lea     rcx, QWORD PTR resource1$[rsp]
call    Resource getResource(void) ; getResource
```

Соглашение такое же, только для передачи указателя используется регистр `rcx`.

Рекомендация

Все основные компиляторы уже много лет назад реализовали оптимизацию именованного и неименованного возвращаемого значения (по моим данным, начиная с Clang 3, GCC 4 и Visual Studio 2008), так что просто возвращайте по значению и не пытайтесь мудрить для «оптимизации» возврата.

Для дополнительного чтения

Статья в блоге 2011 года, демонстрирующая, что основные компиляторы уже прекрасно справляются с пропуском копирования
https://blog.knatten.org/2011/08/26/dont-be-afraid-of-returning-by-value-know-the-return-valueoptimization/.

Гарантированный пропуск копирования не пропускает копирование
https://blog.tartanllama.xyz/guaranteed-copy-elision/.

Пропуск копирования и оптимизация возвращаемого значения
https://en.cppreference.com/w/cpp/language/copy_elision.

Itanium C++ ABI
https://itanium-cxx-abi.github.io/cxx-abi/abi.html.

Объяснение ассемблерного кода в этом примере
https://www.youtube.com/watch?v=I7j6QCO8xMc.

Задача 10

Странное присваивание

a-strange-assignment.cpp

```
#include <iostream>
#include <string>

std::string getName()
{
    return "Alice";
}
int main()
{
    std::string name{"Bob"};
    getName() = name;
    std::cout << "Assigned to a function!\n";
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

Программа напечатает следующий результат:

```
Assigned to a function!
```

Обсуждение

Что, что?! Мы можем присвоить функции? Да что это вообще значит?

На первый взгляд, мы присваиваем переменную вызову функции, хотя обычно видим нечто прямо противоположное (`name = getName()`). Давайте посмотрим внимательнее, что тут происходит.

Выражение `getName()` вызывает функцию `getName`, которая возвращает строку `"Alice"`. Это приводит к созданию временного объекта типа `std::string` со значением `"Alice"`. Затем мы выполняем копирующее присваивание переменной `name` (которая содержит значение `"Bob"`) этому временному объекту. При таком взгляде все обретает смысл. Но при ближайшем рассмотрении возникает ряд вопросов.

В задаче 9 мы познакомились с pr-значениями:

```
Resource getResource() { return Resource{}; }
Resource resource1 = getResource();
```

Там мы выяснили, что вызов функции `getResource()` является pr-значением и не приводит к созданию временного объекта `Resource`. Вместо этого возвращаемое значение создается непосредственно в объекте `resource1`, и никакого копирования не производится.

Выражение `getName()` в этой задаче тоже является pr-значением, поэтому возвращенный объект даже не существует до тех пор, пока мы не присвоим его чему-то, а как раз этого мы и не делаем!

Но если объекта не существует, то как можно что-то присвоить ему копированием? На помощь приходит материализация временных объектов. Причина в том, что вызовы функций являются pr-значениями и не создают объектов вовсе, чтобы избежать лишнего копирования. В примере `Resource` выше создание настоящего объекта откладывается до того момента, как мы присваиваем результат `getResource()` переменной `resource1`. Но иногда мы вызываем функцию только ради ее побочных эффектов и не присваиваем результат ничему. Именно так происходит с вызовом `getName()` в этой задаче. Если результат выражения не присваивается ничему, то мы говорим, что это выражение с отбрасываемым значением. Но мы все равно хотим, чтобы объект был создан, так что временный объект

материализуется. Здесь мы это и наблюдаем – pr-значение `getName()` создает временный объект типа `std::string`.

Теперь, имея настоящий объект `std::string`, мы можем присвоить ему `name` копированием.

Привязка ссылок к pr-значениям

Еще один случай, когда материализация временных объектов важна, – привязка ссылки к pr-значению. Допустим, к примеру, что имеется функция одного из следующих типов:

```
void useName(const std::string& name_ref);
void useName(std::string&& name_ref);
```

Если вызвать `useName(getName());`, то для ссылочных параметров `name_ref` должны существовать настоящие объекты, к которым можно привязаться. Так и происходит, когда ссылка привязывается к pr-значению, – временный объект материализуется.

Для дополнительного чтения

Lvalues, Rvalues, Glvalues, Prvalues, Xvalues, Help!
https://blog.knatten.org/2018/03/09/lvalues-rvalues-glvalues-prvalues-xvalues-help/.

Задача 11

Когда наступила смерть?

whats-the-time-of-death.cpp

```cpp
#include <iostream>

struct MemoryArea
{
    MemoryArea(int number) : number_(number) {}
    ~MemoryArea() {
        std::cout << "Freed memory area " << number_ << "\n";
    }
    int number_;
};
MemoryArea getMemory(int number) { return MemoryArea{number}; }
struct DataSource
{
    DataSource(const MemoryArea &memoryArea)
        : memoryArea_(memoryArea) {}
    const MemoryArea &memoryArea_;
};
int main()
{
    const auto &reference1 = getMemory(1);
    std::cout << "Bound reference 1\n";
    const auto &reference2 = getMemory(2).number_;
    std::cout << "Bound reference 2\n";
    const auto &reference3 = DataSource(getMemory(3));
    std::cout << "Bound reference 3\n";
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

Программа напечатает следующий результат:

```
Bound reference 1
Bound reference 2
Freed memory area 3
Bound reference 3
Freed memory area 2
Freed memory area 1
```

Обсуждение

Мы уже несколько раз видели, что вызов функции – это pr-значение, и никакого объекта не существует, пока это pr-значение не будет использовано для его инициализации. В предыдущей задаче мы также видели исключение из этого правила: ситуацию, когда pr-значение не использовалось для инициализации объекта, мы назвали выражением с игнорируемым значением и выяснили, что тогда временный объект материализуется.

В этой задаче результаты вычисления выражений вызова функций не присваиваются объектам, как в задаче о подсчете копий, но и не отбрасываются, как в задаче о странном присваивании. Вместо этого мы связываем с ними ссылки различными способами. И что тогда происходит?

- В первом вызове мы связываем ссылку `reference1` непосредственно с pr-значением. При этом материализуется временный объект, о чем мы вскользь говорили в задаче 10.
- Во втором вызове мы обращаемся к члену `number_` объекта, являющегося pr-значением. Это еще один случай, когда необходимо материализовать временный объект. Иначе не будет никакого объекта, к члену которого мы могли бы обратиться.
- В третьем вызове мы передаем pr-значение конструктору `DataSource`. Это фактически тот же случай, что в первом вызове; ссылочный параметр конструктора `memoryArea` привязывается к pr-значению, и материализуется временный объект.

Итак, временный объект материализуется при каждом из трех вызовов `getMemory`. Но как долго они остаются в живых?

Обычно временный объект уничтожается в конце полного выражения, в котором был создан. В случаях, описанных в этой задаче, это означает, что они будут уничтожены в конце строки. Однако иногда время жизни временного объекта продлевается. В какой-то мере это пережиток прошлого, ныне уже не очень полезный. Напри-

мер, вместо `const auto &reference1 = getMemory(1);` гораздо лучше было бы написать просто `const auto object1 = getMemory(1);`. Благодаря новым правилам обращения с pr-значениями в C++17 лишнего копирования не будет, и нам не придется задавать все эти вопросы. А чем меньше вопросов, тем лучше код!

Так как же формулируются правила продления времени жизни? Их несколько, но самые употребительные и интересующие нас в данном случае – следующие два:

- если вы связываете ссылку с материализованным временным объектом, то этот временный объект существует на протяжении времени жизни ссылки;
- если вы связываете ссылку с *членом* материализованного временного объекта, то весь этот временный объект (а не только его член) существует на протяжении времени жизни ссылки.

Рассмотрим каждую строку кода в текущей задаче.

```
const auto &reference1 = getMemory(1);
```

Мы связываем ссылку `reference1` с временным объектом, полученным от `getMemory(1)`. Временный объект 1 существует на протяжении времени жизни `reference1`, т. е. до конца `main`.

```
std::cout << "Bound reference 1\n";
```

Печатается сообщение `Bound reference 1`.

```
const auto &reference2 = getMemory(2).number_;
```

Мы связываем ссылку `reference2` с членом `number_` временного объекта, полученного от `getMemory(2)`. Полный временный объект 2 (а не только его член) существует на протяжении времени жизни `reference2`, т. е. до конца `main`.

```
std::cout << "Bound reference 2\n";
```

Печатается сообщение `Bound reference 2`.

```
const auto &reference3 = DataSource(getMemory(3));
```

Мы связываем ссылочный параметр конструктора `memoryArea` с временным объектом, полученным от `getMemory(3)`. Он используется для создания нового объекта типа `DataSource`. Затем мы связываем ссылку `reference3` с временным объектом `DataSource`, материализовавшимся при вызове его конструктора. Временный объект `DataSource` существует на протяжении времени жизни `reference3`, т. е. до конца `main`.

Стоп, это засада! Как только объект `DataSource` был сконструирован, параметр `memoryArea` пропал. Не помогает и то, что мы присвоили его члену `memoryArea_`, – это другая ссылка! Следовательно, время жизни третьего временного объекта `MemoryArea` *не* продлевается, и память освобождается в конце строки. Печатается сообщение `Freed memory area 3`, и значит, область `MemoryArea`, из которой `DataSource` собирался читать, уже освобождена, и теперь мы имеем висячую ссылку.

```
std::cout << "Bound reference 3\n";
```

Печатается сообщение `Bound reference 3`.

```
}
```

Ах, эта `}`, лучшее, что есть в C++! В конце области видимости локальные переменные уничтожаются в порядке, обратном порядку их конструирования. Сначала уничтожается `reference3` и, стало быть, временный объект `DataSource` (его деструктор не печатает ничего). Затем уничтожается `reference2,` а с ней и временный объект 2, и печатается `Freed memory 2`. Наконец, уничтожается `reference1`, а значит, и временный объект 1, и печатается `Freed memory 1`.

Смежные вопросы

Проблемы, схожие с той, что возникает для временного объекта 3, не редкость в C++. Вот знаменитый пример, который не был исправлен до выхода стандарта C++23. (На момент написания книги основным компиляторам еще только предстояло реализовать это исправление.)

examples/range-for/range-for.cpp

```
class MemoryAreaContainer
{
    public:
        MemoryAreaContainer();
        std::vector<MemoryArea> &getMemoryAreas()
        {
            return memoryAreas_;
        }
    private:
        std::vector<MemoryArea> memoryAreas_;
};
```

examples/range-for/range-for.cpp

```
for (const auto &lp : MemoryAreaContainer{}.getMemoryAreas())
{
      std::cout << lp.number_ << std::endl;
}
```

Выглядит безобидно. Но чтобы временный объект `MemoryAreaContainer` существовал до конца цикла for, мы должны связать с ним константную ссылку. Здесь наша константная ссылка связана не с `MemoryAreaContainer`, а со значением, возвращенным из `getMemoryAreas()`. Поэтому временный объект `MemoryAreaContainer` уничтожается еще до того, как мы начнем итерации. Ссылка, полученная от `getMemoryAreas()`, ссылается на уже уничтоженный член `memoryAreas_` и является висячей. (Это немного упрощенное изложение; полный код, в который раскрывается основанный на диапазоне цикл for, см. в разделе стандарта stmt.ranged[1].)

Рекомендации

- Не полагайтесь на продление времени жизни, предпочитайте форму `const auto obj = get()` форме `const auto& obj = get()`.
- Не полагайтесь на продление времени жизни за пределы вызова функции.
- Не отставайте от стандартов C++, компиляторов и инструментов. Часто в компиляторах и даже в самом языке исправляются ошибки.
- Пользуйтесь контролерами. В частности, контролер адресов обнаруживает некоторые случаи использования уже уничтоженных объектов.

1 *https://timsong-cpp.github.io/cppwp/std20/stmt.ranged*.

Для дополнительного чтения

Время жизни временного объекта
https://en.cppreference.com/w/cpp/language/reference_initialization#Lifetime_of_a_temporary.

Окончательное исправление основанного на диапазоне цикла for
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2644r1.pdf.

Задача 12

Фальстарт

a-false-start.cpp

```cpp
#include <iostream>
#include <stdexcept>
struct Engine
{
    ~Engine() { std::cout << "Engine stopped\n"; }
};
struct Machine
{
    Machine() { throw std::runtime_error{"Failed to start machine"}; }
    ~Machine() { std::cout << "Machine stopped\n"; }
    Engine engine_;
};
int main()
{
    try
    {
        Machine machine;
    }
    catch (...)
    {}
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

Программа напечатает следующий результат:

```
Engine stopped
```

Обсуждение

В этой задаче имеется класс `Machine` и в нем член типа `Engine`. Но `Machine` возбуждает исключение в конструкторе еще до завершения инициализации. Остановится ли двигатель? Остановится ли сама машина?

Одна из замечательных особенностей C++ заключается в том, что сконструированные объекты уничтожаются автоматически и детерминированно (если только вы не вызываете вручную `new`, что в современных реалиях следует оставить авторам библиотек). Не возникает никакой необходимости в конструкции `with` из Python или `using` из C#, и даже потребность в чем-то типа `finally` возникает редко.

Но уничтожаются только сконструированные объекты, и возникает вопрос: а что такое «быть сконструированным»? И что происходит в случае таких «наполовину сконструированных» объектов, как экземпляр класса `Machine` выше, которые возбудили исключение во время конструирования?

Объект считается сконструированным, только когда его инициализация завершена. Инициализация объекта класса производится в следующем порядке:

- виртуальные базовые классы;
- прямые базовые классы;
- нестатические члены данных (такие как `Machine::engine_`);
- тело конструктора.

Итак, при конструировании объекта `Machine` мы сначала инициализируем член типа `Engine`. Сразу после этого `engine_` считается сконструированным и в конечном итоге должен быть уничтожен. Затем мы входим в тело конструктора `Machine`, которое возбуждает исключение. Инициализация `Machine` не завершена, и он не считается сконструированным. Будет вызван только деструктор `Engine`, но не деструктор `Machine`.

Это очень полезно, поскольку вам не нужно думать об очистке наполовину инициализированных объектов. Любой инициализированный член, такой как `engine_`, уничтожается автоматически; остальные не нужно уничтожать, потому что они никогда и не были сконструированы. Однако необходима осторожность, если вы вручную захватываете какой-то ресурс в конструкторе. В этом случае

нельзя полагаться на то, что деструктор его освободит. Например, если вы вызываете `new` и присваиваете указатель члену, то не можете рассчитывать на то, что `delete` в вашем деструкторе произведет очистку. То же самое относится к парам низкоуровневых функций типа `open`/`close`.

Как всегда, чтобы сделать код надежнее, избегайте вручную вызывать такие функции, как `delete` и `close`. Вместо этого пользуйтесь для определения членов типами из стандартной библиотеки, например `unique_ptr` или `fstream`, которые автоматически освобождают управляемые ресурсы в своих деструкторах. Если имеется несколько таких членов-ресурсов, то те, которые были инициализированы, и только они будут автоматически освобождены даже в случае, когда объект был сконструирован лишь частично, как `machine`.

Если ваш класс управляет специальным ресурсом, для которого нет никакого типа в стандартной библиотеке, как, например, двигатель (`engine_`) в этой задаче, который необходимо остановить, то вынесите эту обязанность в отдельный небольшой класс, скажем `Engine`, а не управляйте им вручную в объемлющем классе. Тогда вам вообще не нужно будет думать об управлении им, он обязательно будет уничтожен, если был сконструирован.

Рекомендации

- Избегайте ручного управления ресурсами, используйте для членов типы из стандартной библиотеки, например `unique_ptr` и `fstream`.
- Для специальных ресурсов создавайте классы, единственная обязанность которых – управление ресурсами этого вида.

Для дополнительного чтения

Порядок инициализации класса

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

Задача 13

Постоянная борьба

a-constant-struggle.cpp

```
#include <iostream>
#include <type_traits>

template <typename T>
void byValue(T t)
{
    std::cout << std::is_const_v<T>; // true, если T – const
}
template <typename T>
void byReference(T &t)
{
    std::cout << std::is_const_v<T>; // true, если T – const
}
int main()
{
    int nonConstInt = 0;
    const int constInt = 0;
    byValue(nonConstInt);
    byValue(constInt);
    byReference(nonConstInt);
    byReference(constInt);
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

Программа напечатает следующий результат:

```
0001
```

Обсуждение

Шаблоны функций `byValue` и `byReference` принимают параметры по значению и по ссылке соответственно. Затем мы вызываем эти шаблоны с неконстантным и константным аргументом типа `int`. В каких случаях `T` выводится как `const`?

Прежде чем переходить к правилам вывода, проясним две концепции, которые часто путают. `T` в шаблоне `<typename T>` называется *параметром шаблона*. Цель вывода аргумента шаблона – найти `T`. С другой стороны, `T` и `T &` в сигнатурах функций `byValue(T t)` и `byReference(T &t)` называются *типами параметра шаблона*. Это фактические типы параметров шаблонов функций, их мы используем для вывода.

Будем обозначать типы параметров шаблонов `P`, чтобы отличить их от параметров шаблонов `T`. Ниже приведено три примера:

	Параметр шаблона	Тип параметра шаблона T
`template <typename T>`		
`void byValue(T t)`	`T`	`T`
`template <typename T>`		
`void byReference(T & t) T T &`	`T`	`T&`
`template <typename ValueType>`		
`void f(const std::vector<ValueType>& v)`	`ValueType`	`const std::vector<ValueType>&`

Как видно из этих примеров, в обеих функциях `byValue` и `byReference` параметром шаблона является `T`. В `byValue` тип параметра шаблона `P` также совпадает с `T`, но в `byReference` он равен `T &`.

Разобравшись с этим, займемся правилами вывода. Сначала посмотрим на вызовы `byValue`:

```
template <typename T> void byValue(T t);

int nonConstInt = 0;
const int constInt = 0;

byValue(nonConstInt);
byValue(constInt);
```

Напомним, что параметр шаблона – `T`, а типом параметра шаблона `P` также является просто `T` (а не, скажем, `T &`).

Если тип параметра шаблона `P` не ссылочный, то правила вывода требуют игнорировать cv-квалификаторы аргумента (`const` или `volatile`) верхнего уровня. Так как `P` – это просто `T`, то тип не ссылочный, и cv-квалификаторы игнорируются.

Стоп, получается, что неважно, передали мы неконстантный или константный аргумент, все равно делаем вид, что он неконстантный! Не странно ли? Нет. Если `P` не ссылочный тип, значит мы передаем аргумент по значению. А тогда даже если аргумент был константным, все равно параметр `t` внутри `byValue` является копией, и модифицировать исходный аргумент мы не можем.

Поскольку cv-квалификаторы верхнего уровня игнорируются, то мы выводим `P` по неконстантному `int` в обоих вызовах `byValue` и находим, что в обоих случаях `P == int`. А так как `P == T`, то `T` также неконстантный. Оба вызова печатают `0`.

Далее рассмотрим обращения к `byReference`:

```
template <typename T> void byReference(T &t);

int nonConstInt = 0;
const int constInt = 0;

byReference(nonConstInt);
byReference(constInt);
```

Параметром шаблона является `T`, а тип параметра шаблона `P` равен `T &`.

На этот раз `P` – ссылка, и мы *не* игнорируем верхнеуровневые cv-квалификаторы аргумента. В этом случае они важны! Ссылочный параметр `t` будет связан с нашим оригинальным аргументом, поэтому важно, чтобы мы не пытались, к примеру, связать неконстантную ссылку с константным аргументом.

Если передан неконстантный аргумент `nonConstInt`, то `P` выводится как `int &`, т. е. `T` – это `int`. Это неконстантный тип, поэтому печатается `0`.

Если передан константный аргумент `constInt`, то `P` выводится как `const int &`, т. е. `T` – это `const int`. Это константный тип, поэтому печатается `1`.

Полные правила вывода аргумента шаблона могут показаться довольно сложными, но, по крайней мере, эта часть, касающаяся игнорирования верхнеуровневых cv-квалификаторов, интуитивно понятна, так что я смог ее запомнить.

Auto

Правила вывода для `auto` основаны на правилах вывода аргумента шаблона. Так что теперь, зная о последних, вы знаете также соответствующие правила вывода для `auto`!

```
auto t1 = nonConstInt;  // неконстантный объект
auto t2 = constInt;     // неконстантный объект
auto &t3 = nonConstInt; // ссылка на неконстантный объект
auto &t4 = constInt;    // ссылка на константный объект
```

Для дополнительного чтения

Вывод аргументов шаблона
https://en.cppreference.com/w/cpp/language/template_argument_deduction.
Вывод аргументов шаблона по вызову функции
https://timsong-cpp.github.io/cppwp/std20/temp.deduct.call.

Задача 14

Аристотелева сумма частей

aristotles-sum-of-parts.cpp

```
#include <iostream>
#include <type_traits>

int main()
{
    char char1 = 1;
    char char2 = 2;

    // True, если char1 + char2 имеет тип char
    std::cout << std::is_same_v<decltype(char1 + char2), char>;
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

Программа напечатает следующий результат:

```
0
```

Обсуждение

Знаю, знаю. Предполагается, что это веселая книжка. Но иногда приходится говорить и на такие грустные темы, как эта.

Как вы уже поняли, тип `char + char`, к сожалению, не `char`. А какой тогда?

Для многих бинарных операций, таких как +, -, *, /, >, ==, ^ и пр., операнды должны иметь одинаковый тип. Но вам самим не нужно это гарантировать, C++ обо всем позаботится. На самом деле C++ с большой охотой занимается неявным преобразованием типом, пока вы не смотрите, и это один из самых путаных случаев. Добро пожаловать в мир арифметических преобразований.

Обычные арифметические преобразования применяются к операндам арифметических операторов с целью привести их к одному типу. Этот общий тип является также типом результата. Например, если вы складываете `float` и `int`, то `int` сначала преобразуется в тип `float`, и результат будет иметь тип `float`. А если вы складываете `float` и `double`, то `float` сначала преобразуется в `double`, и результат будет иметь тип `double`. Пока что все разумно.

Путаница начинается, когда оба операнда имеют целый тип, особенно если этот тип *один и тот же*. Например, если вы складываете `char` и `int`, то `char` сначала преобразуется в `int`, и результат будет иметь тип `int`. Это тоже понятно. Но что, если складываются `char` и `short`? Вместо того чтобы преобразовать `char` в `short`, оба операнда преобразуются к типу `int` (обычно – но читайте дальше). Хуже того – если складываются два `char`, т. е. оба операнда имеют одинаковые типы, то они все равно преобразуются к другому типу!

Посмотрим, что обычные арифметические преобразования делают с нашим выражением `char1 + char2`. Поскольку оба операнда целого типа, к обоим применяется процесс, называемый целочисленным расширением. Идея в том, чтобы преобразовать более узкий целый тип в `int` или `unsigned int` до вычисления арифметического выражения. После преобразования обоих операндов мы можем сложить их, выполнив обычное сложение `int` или `unsigned int`, и получить в качестве результата значение типа `int` или `unsigned int`.

Правило расширения более узкого целого типа заключается в том, что его можно преобразовать в `int`, если `int` способен пред-

ставить все значения более узкого типа; в противном случае его можно преобразовать в `unsigned int`. Верно ли, что `char` помещается в `int`? Обычно да, но это зависит от реализации. (Я же говорил, что это грустная история.)

Осложняет дело то, что размеры фундаментальных типов в C++ не фиксированы. Реализация может решить, что ширина `int` составляет 16 или 64 бит и даже что `char` и `long` имеют одинаковый размер. Правила всего два:

- каждый из типов `signed char`, `short`, `int`, `long` и `long long` должен быть не у́же предыдущего;
- для каждого типа в этом списке существует вариант `unsigned`, имеющий такой же размер как вариант `signed`.

Также определены некоторые минимальные размеры:

Тип	Минимальная ширина
`signed char`	8
`short`	16
`int`	16
`long`	32
`long long`	64

В типичной 64-разрядной системе Linux, Windows или macOS тип `int` 32-разрядный, а тип `long` 64-разрядный в Linux/macOS и 32-разрядный в Windows. Но отвечающая стандарту реализация с тем же успехом вправе решить, что все целые типы должны быть 64-разрядными.

В этой задаче нас интересует конкретно тип `char`, но именно он почему-то отсутствует в приведенных выше списках. В чем дело? Тип `char` особый в том смысле, что реализация вправе решить, совпадает он с `signed char` или `unsigned char`. В любом случае `char` – самостоятельный тип. Таким образом, `char`, `signed char` и `unsigned char` – три разных типа, а является просто `char` типом со знаком или без знака, решает реализация!

Напомним, что нас интересует, все ли значения типа `char` умещаются в тип `int`, – это позволит определить, будут наши переменные типа `char` преобразованы в `int` или в `unsigned int`. Обычно они умещаются, так что оба операнда `char1 + char2` будут преобразованы в `int`, и результат тоже будет иметь тип `int`.

Но предположим, что реализация решила, что оба типа `char` и `int` будут 16-разрядными. Предположим также, что реализация постулировала, что `char` – беззнаковый тип. Тогда диапазон `int` будет от -2^{16-1} до $2^{16-1} - 1$, или от –32768 до 32767 включительно. Диапазон `char` будет от 0 до $2^{16} - 1$, или от 0 до 65535 включительно. То есть половина `char` не помещается в `int`! В таких системах операнды `char1 + char2` будут расширены до `unsigned int`, и результат тоже будет иметь тип `unsigned int`.

Таким образом, `char1 + char2` может иметь тип `int` или `unsigned int`. Но мы точно знаем, что это не `char`.

Для дополнительного чтения

Обычная путаница с арифметикой
https://shafik.github.io/c++/2021/12/30/usual_arithmetic_confusions.html.

64-разрядные модели данных
https://en.wikipedia.org/wiki/64-bit_computing#64-bit_data_models.

Фундаментальные типы
https://timsong-cpp.github.io/cppwp/std20/basic.fundamental.

Использование арифметических преобразований
https://timsong-cpp.github.io/cppwp/std20/expr.arith.conv.

Целочисленные расширения
https://timsong-cpp.github.io/cppwp/std20/conv.prom.

Задача 15

Назад из будущего

back-from-the-future.cpp

```
#include <future>
#include <iostream>

int main()
{
    char counter = 0;
    auto future1 = std::async(std::launch::async, [&]()
    {
        counter++;
    });
    auto future2 = std::async(std::launch::async, [&]()
    {
        return counter;
    });
    future1.wait();
    // Привести к int, чтобы напечатать как числовое значение, а не символ
    std::cout << (int)future2.get();
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

Поведение программы не определено! Важно, что ответ не «0 или 1».

Обсуждение

Когда два несинхронизированных потока обращаются к одной и той же ячейке памяти и хотя бы одно из этих обращений является модификацией, возникает гонка за данные. Эта программа запускает два потока, один из которых модифицирует переменную `counter`, а другой читает ее. Поскольку мы ничего не сделали для синхронизации потоков, в программе имеет место гонка за данные.

Если вы запустите эту программу на своей машине, то, скорее всего, увидите `0` или `1` в зависимости от того, какой поток доберется до `counter` первым. Переменная `counter` занимает один байт, поэтому вы вряд ли увидите наполовину записанное значение. Либо операция `counter++` была выполнена целиком до того, как второй поток прочитал `counter`, либо после – но тоже целиком. Но это не значит, что можно полагаться на такую «непротиворечивую в конечном счете» логику! Гонка за данные – это неопределенное поведение, поэтому произойти может все что угодно. Вспомните, о чем мы говорили в предисловии, – у вас из носа могут вылететь демоны. Не пытайтесь рассуждать о том, что произойдет при неопределенном поведении. И тестировать это тоже не стоит. Может казаться, что ваша программа работает, но не исключено, что она перестанет работать в другой системе или после обновления компилятора или на каждом миллионном прогоне.

Чтобы эти потоки могли безопасно обращаться к `counter`, необходима какая-то синхронизация. Классическое решение – защитить `counter` мьютексом, но это громоздко и требует добавления блокировок. Блокировки могут работать медленно, а кроме того, когда есть блокировки, возможны и взаимоблокировки. Лучше воспользоваться атомарным типом данных:

examples/std_atomic/std_atomic.cpp

```
std::atomic<char> counter = 0;

auto future1 = std::async(std::launch::async, [&]() {
                              counter++;
                          });
```

```
auto future2 = std::async(std::launch::async, [&]() {
                            return counter.load();
                         });
```

Использование std::atomic вместо мьютекса делает код проще, позволяет избежать заведения дополнительной переменной для мьютекса, гарантирует, что мы не забудем захватить мьютекс, и, скорее всего, избавляет код от взаимоблокировок.

Безблокировочные атомарные типы

Действительно ли тип std::atomic<char> безблокировочный, зависит от реализации, но обычно так оно и есть. Вы можете проверить, так ли это в вашей системе, воспользовавшись членами is_lock_free или is_always_lock_free шаблона класса std::atomic:

examples/std_atomic/std_atomic.cpp

```
static_assert(counter.is_always_lock_free);
assert(counter.is_lock_free());
```

- is_always_lock_free возвращает true, если std::atomic<T> для данного T всегда безблокировочный. Это член вида static constexpr, который можно проверить на этапе компиляции.
- is_lock_free возвращает true, если *конкретный* экземпляр std::atomic<T> безблокировочный. Например, некоторые атомарные типы являются безблокировочными, только если выровнены на ту или иную границу в памяти. Это можно проверить только на этапе компиляции.

На моих компьютерах M1 Macbook и x86_64 Linux шаблон std::atomic с параметрами int, long и double всегда безблокировочный.

Прекрасный инструмент для обнаружения гонок за данные – бесплатный контролер ThreadSanitizer. Это один из многих контролеров, входящих в комплект LLVM, а для его использования нужно просто откомпилировать программу с параметром -fsanitize=thread, а затем запустить ее. Для рассматриваемой задачи будет напечатано что-то типа:

```
WARNING: ThreadSanitizer: data race (pid=2735)
  Read of size 1 at 0x7ffed85a35cf by thread T2:
    (... трасса стека ...)
Previous write of size 1 at 0x7ffed85a35cf by thread T1:
    (... трасса стека ...)
Location is stack of main thread.
Location is global '??' at 0x7ffed8584000 ([stack]+0x1f5cf)
Thread T2 (tid=2738, running) created by main thread at:
    (... трасса стека ...)
Thread T1 (tid=2737, finished) created by main thread at:
    (... трасса стека ...)
```

Как видите, контролер не только сообщает, что имела место гонка за данные, но и где находится переменная, трассы стека до места, где она читается и записывается и до места, где создаются потоки. Полезная штучка!

Рекомендации

- Всегда синхронизируйте операции доступа к разделяемым переменным, если хотя бы одна из них является записью.
- Шаблон `std::atomic` часто является отличным выбором для простой синхронизации значений.
- Пользуйтесь контролерами.

Для дополнительного чтения

std::async
https://en.cppreference.com/w/cpp/thread/async.

std::atomic
https://en.cppreference.com/w/cpp/atomic/atomic.

Определение гонок за данные в стандарте C++
https://timsong-cpp.github.io/cppwp/std20/intro.races.

ThreadSanitizer
https://clang.llvm.org/docs/ThreadSanitizer.html.

Задача 16

Перегруженный контейнер

an-overloaded-container.cpp

```
#include <initializer_list>
#include <iostream>

struct Container
{
    Container(int, int)
    {
        std::cout << "Two ints\n";
    }
    Container(std::initializer_list<float>)
    {
        std::cout << "std::initializer_list<float>\n";
    }
};
int main()
{
    Container container1(1, 2);
    Container container2{1, 2};
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

Программа напечатает:

```
Two ints
std::initializer_list<float>
```

Дополнительный вопрос: что случится, если убрать конструктор с параметром типа `initializer_list`? Откомпилируется ли программа?

Обсуждение

В структуре `Container` есть два конструктора: один принимает два `int`, другой – `std::initializer_list<float>`. Конструируются два объекта типа `Container`, и оба получают два `int` в качестве аргументов. Но есть и разница: в первом случае параметры передаются в круглых скобках, `(1,2)`, а во втором – в фигурных, `{1,2}`. Какой конструктор вызывается в каждом случае?

Для `Container container1(1, 2)` процедура разрешения перегрузки выбирает конструктор, принимающий два `int`, что и неудивительно.

Для `Container container2{1, 2}` процедура разрешения перегрузки выбирает конструктор, принимающий `std::initialization<float>`, хотя мы и передали два `int`! Означает ли это, что `{1,2}` – объект типа `std::initializer_list`? Нет, `{1,2}` – это не объект типа `std::initializer_list`, а выражение, называемое *списком инициализации в фигурных скобках* (braced-init-list). Оно состоит из нуля или более элементов, заключенных в фигурные скобки. Это допускается в нескольких контекстах, из которых инициализация самый распространенный. Вот ряд примеров использования списков инициализации в фигурных скобках для инициализации:

examples/braced-init-list/braced-init-list.cpp

```
std::array<int, 3> a{1, 2, 3};
int i{2};
int j{};
f({1, 2});
std::map<int, std::string>{{1, "one"}, {2, "two"}};
return {1, 2};
```

Как видно из примеров инициализации `i` и `j`, даже если внутри фигурных скобок находится нуль или один элемент, а не «список» элементов, выражение все равно называется списком инициализации в фигурных скобках.

Инициализация с помощью списка инициализации в фигурных скобках называется *инициализацией списком* (list-initialization), а сам этот список в этих контекстах неудачно называется *списком инициализаторов* (initializer list). Список инициализаторов не то же самое, что шаблон класса `std::initializer_list<T>`. Эти имена только вносят путаницу.

Итак, если `{1,2}` не `std::initializer_list`, то почему выбирается конструктор `std::initializer_list`? В правилах разрешения перегрузки есть специальный случай для инициализации списком, состоящий из двух этапов:

- сначала производится разрешение перегрузки для всех конструкторов, принимающих `std::initializer_list`, когда список инициализации в скобках рассматривается как один аргумент;
- затем и только в случае, когда первый этап не увенчался успехом, производится разрешение перегрузки для остальных конструкторов, когда элементы списка инициализации в скобках рассматриваются как отдельные аргументы.

Таким образом, в случае `Container container2{1, 2}` процедура разрешения перегрузки сначала пробует конструктор `std::initializer_list<float>`. Тип `int` можно неявно преобразовать в тип `float`, и аналогично список инициализаторов типа `int` можно использовать для создания `std::initializer_list<float>`. Поэтому выбирается конструктор `std::initializer_list<float>`, а второй конструктор даже не рассматривается.

Теперь перейдем к дополнительному вопросу: что, если удалить конструктор `std::initializer_list<float>`? Для `Container container1(1, 2)` никаких изменений не будет, потому что это не инициализация списком. А для `Container container2{1, 2}` процедура разрешения перегрузки не найдет ни одного конструктора, принимающего `std::initializer_list`, на первом этапе и перейдет к рассмотрению второго конструктора. Стало быть, конструктор, принимающий два `int`, используется в обоих случаях, и программа компилируется.

std::vector

Если вы когда-нибудь задавались вопросом, почему `std::vector<int> v(1,2)` дает вектор с одним элементом, `2`, а `std::vector<int> v{1,2}` – вектор с двумя элементами, `1` и `2`, то теперь вы знаете ответ:

- в первом случае нет никакой инициализации списком, а выбирается конструктор, которые принимает аргументы `size_type count` и `T value`;
- во втором случае имеет место инициализация списком, и выбирается конструктор, принимающий `std::initializer_list<T>`.

Для дополнительного чтения

Инициализация списком
https://en.cppreference.com/w/cpp/language/list_initialization.

std::initializer_list
https://en.cppreference.com/w/cpp/utility/initializer_list.

Конструкторы std::vector:
https://en.cppreference.com/w/cpp/container/vector/vector.

Задача 17

Строгий указ

a-strong-point.cpp

```
#include <iostream>

struct Points
{
    Points(int value) : value_(value) {}
    int value_;
};
struct Player
{
    explicit Player(Points points) : points_(points) {}
    Points points_;
};
int main()
{
    Player player(3);
    std::cout << player.points_.value_;
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

Программа печатает следующий результат:

```
3
```

Обсуждение

В структуре `Player` имеется член `points_`, в котором хранится количество набранных игроком очков. Вместо фундаментального типа, например `int`, мы решили использовать пользовательский тип `Points`. Такая техника часто называется «строгой типизацией», благодаря ей код проще читать, и ошибки менее вероятны. Например, если имеется функция `do_something(int player_id, int points, int lives)`, то легко перепутать порядок параметров и передать `player_id, lives, points`, т. е. количество жизней вместо очков и наоборот. Когда для этих значений используются строгие типы, подобный казус воспрепятствует успешной компиляции, но зато в программу не будет внесена ошибка.

Вопрос в задаче состоит в том, разрешено ли конструировать объект `Player` непосредственно с параметром типа `int`, как в выражении `Player player(3)`, или нужно вручную создать объект строгого типа `Points` и передать его конструктору `Player`.

Чтобы объявление `Player player(3)` работало, нам необходимо иметь неявное преобразование из `int`. А в неявных преобразованиях могут принимать участие только конструкторы, не помеченные ключевым словом `explicit` (они называются преобразующими конструкторами). Конструктор `Player` помечен как `explicit` и, стало быть, не является преобразующим, тогда как же работает это преобразование?

Форма `Player player(3)` – пример *прямой инициализации* (direct-initialization) (как и форма `Player player{3}`, но не, к примеру, `Player player = 3`). Я воспринимаю этот термин как *прямой* вызов конструктора `Player`. При прямой инициализации для выбора наилучшего конструктора применяется разрешение перегрузки. Когда мы прямо инициализируем `Player` из `int`, процедура разрешения перегрузки пытается найти последовательность неявных преобразований из `int` в каждый из конструкторов `Player`. В структуре `Player` есть только один конструктор, и он принимает `Points`, поэтому нам нужно неявное преобразование из `int` в `Points`. Конструктор `Points`

не является явным, а значит, это преобразующий конструктор, и его можно использовать в последовательности неявных преобразований. Число `3` неявно преобразуется в `Points`, и этот объект `Points` передается конструктору `Player`.

Сам конструктор `Player` не принимает участия в неявном преобразовании; помните, мы вызываем его напрямую. Поэтому не имеет значения, явный он или нет.

С другой стороны, если бы конструктор `Points` был явным, то он перестал бы быть преобразующим, и использовать его в последовательности неявных преобразований было бы нельзя. Тогда мы не смогли бы преобразовать `3` в `Points`, и программа не откомпилировалась бы.

Включение в строгие типы только явных конструкторов делает код чуть более многословным, но зато и более надежным. Возвращаясь к примеру функции `do_something`, вы тогда должны были бы написать `do_something(PlayerId{42}, Points{3}, Lives{1})`, чтобы программа откомпилировалась, тогда как при наличии преобразующих конструкторов можно было бы полениться и написать `do_something(42, 1, 3)` – программа все равно откомпилируется. Лично мне первая форма кажется более понятной.

Инициализация копированием и прямая инициализация

Что будет, если написать

```
Player player = 3
```

Эта форма называется *инициализацией копированием* (copy-initialization) и несколько отличается от прямой инициализации. Теперь мы уже не вызываем конструктор `Player` напрямую, а вместо этого пытаемся неявно преобразовать `int` в `Player`. В последовательности неявных преобразований может встречаться только *одно* пользовательское преобразование, но нам понадобилось бы одно для преобразования `int` в `Player` и другое для преобразования `Player` в `Points`. Таким образом, объявление `Player player = 3` не откомпилировалось бы вне зависимости от того, есть среди конструкторов явные или нет.

Рекомендации

- Помечайте конструкторы с одним аргументом ключевым словом `explicit`, чтобы избежать неожиданных неявных преобразований.
- Используйте строгие типы вместо фундаментальных, чтобы избежать ошибок.
- Подумайте об использовании библиотеки строгих типов, например `strong_type` Бьярна Фаллера[1], чтобы уменьшить объем стереотипного кода при создании строгих типов.

Для дополнительного чтения

Прямая инициализация
https://en.cppreference.com/w/cpp/language/direct_initialization.

Инициализация копированием:
https://en.cppreference.com/w/cpp/language/copy_initialization.

Основные принципы C++ C.46: по умолчанию объявляйте конструкторы с одним аргументом явными:
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c46-by-default-declare-singleargument-constructors-explicit.

1 *https://github.com/rollbear/strong_type*.

Задача 18

Освобождаем помещение

moving-out.cpp

```
#include <iostream>
#include <string>

int main()
{
    std::string hello{"Hello, World!"};
    std::string other(std::move(hello));
    std::cout << "'" << hello << "'";
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

У этой программы неспецифицированное поведение! Но, вероятно, она напечатает следующий результат:

```
''
```

Обсуждение

Перемещенный объект больше не должен использоваться, поскольку ресурсы из него «вывезены», указатели могут быть направлены на уничтоженные объекты и т. д.

Как ведет себя конкретный перемещенный объект, зависит от автора класса. Тут у автора почти полная свобода, но как минимум уничтожение объекта должно быть безопасным, чтобы ничего не «сломалось», когда, например, перемещенная локальная переменная выходит из области видимости. Если иное явно не документировано, пользователь не вправе предполагать, что класс дает еще какие-то гарантии.

Впрочем, стандарт C++ дает дополнительную гарантию для классов в стандартной библиотеке:

> Из объектов тех типов, которые определены в стандартной библиотеке, разрешено перемещение. <...> Если явно не оговорено противное, то такие перемещенные объекты переводятся в допустимое, но неспецифицированное состояние.

Для `std::string` противное не оговорено, поэтому перемещенная строка переводится в «допустимое, но неспецифицированное состояние». Но что это значит?

Стандарт C++ дает следующее определение:

> (a) значение объекта не определено, известно только, что все инварианты объекта удовлетворяются, и его операции ведут себя так, как описано в его типе.

Итак, хотя нам и неизвестно, что содержит перемещенная строка, мы, по крайней мере, знаем, что `.size()` возвращает размер того, что в ней осталось, что `.empty()` возвращает `true` тогда и только тогда, когда `.size() == 0` и т. д. Но тем не менее истинное состояние объекта не специфицировано, и мы не можем быть уверены, что выведет программа, поэтому лучше вообще не использовать перемещенные объекты.

Реализация перемещающего конструктора `std::string`

Обычно перемещающий конструктор `std::string` реализуется так, что новая строка «крадет» память у старой, чтобы избежать ее копирования. Тогда важно, чтобы нельзя было обратиться к той же памяти через объект старой строки, иначе модификация одной строки отразилась бы на другой. Важно также, чтобы старая строка не пыталась освободить эту память в своем деструкторе.

Если в реализации `std::string` используется указатель на буфер строки в куче, то можно, например, представить себе, что перемещающий конструктор записывает `null` в указатель на буфер в старой строке. Аналогичную технику вы можете использовать в собственных классах.

В некоторых реализациях короткие строки оптимизируются путем хранения строки в самом объекте `std::string` вместо выделения буфера в куче; такая техника называется оптимизацией коротких строк. В этом случае перемещающий конструктор не может украсть память старой строки, и необходимо копирование. Для такой перемещенной строки было бы разумно оставить оригинальную строку – состояние вполне допустимое.

Я проверял это в Windows, Mac и Linux как для длинных, так и для коротких строк. Во всех случаях печаталась пустая строка `‘’`. Но, как при любом неспецифицированном поведении, я не стал бы предполагать, что так будет всегда. Я только предполагаю, что все инварианты `std::string` удовлетворяются.

Рекомендации

- Не используйте перемещенные объекты (их можно только уничтожить).
- В своих классах старайтесь писать перемещающие конструкторы и перемещающие операторы присваивания так, чтобы они переводили объект в допустимое состояние, в котором удовлетворяются все инварианты объекта.
- Если так сделать не получается, то, по крайней мере, гарантируйте, что вызов деструктора безопасен.

Для дополнительного чтения

Конструкторы std::string
https://en.cppreference.com/w/cpp/string/basic_string/basic_string.

Перемещенное состояние для библиотечных типов
https://timsong-cpp.github.io/cppwp/std20/lib.types.movedfrom.

Допустимое, но неспецифицированное состояние
https://timsong-cpp.github.io/cppwp/std20/defns.valid.

Задача 19

Маленькая сумма

a-little-sum-thing.cpp

```
#include <iostream>
#include <numeric>
#include <vector>

int main()
{
    std::vector<int> v{-2, -3};
    std::cout << std::accumulate(v.cbegin(), v.cend(), v.size());
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

У этой программы зависящее от реализации поведение! Но она напечатает большое число, например такое:

```
18446744073709551613
```

Обсуждение

Размер вектора равен 2. Почему после прибавления к нему -2 и -3 получается не -3, а что-то другое?

Шаблон функции std::accumulate определен следующим образом:

```
template<class InputIterator, class T>
constexpr T accumulate(InputIterator first, InputIterator last, T init);
```

> Эффекты: вычисляет результат путем инициализации аккумулятора acc начальным значением, а затем модифицирует его путем выполнения операции acc = std::move(acc) + *i (…) для каждого итератора i в диапазоне [first, last) по порядку.

Начнем с вывода параметров шаблона. Результатом вывода InputIterator будет тип итератора для вектора, который в этой задаче не представляет интереса, потому что используется только для итерирования. С другой стороны, вывод T даст тип std::vector<int>::size(), что и является ключом к этой задаче. Функция std::vector<int>::size() возвращает значение типа size_type, который в стандарте детально не определен. Требуется только, чтобы это был целый тип без знака, и, коль скоро это требование выполнено, реализация может выбирать наиболее естественное представление для конкретной системы.

Таким образом, и типом init, и типом возвращаемого значения, а равно и аккумулятора acc, который скрыт внутри std::accumulate, будет целый тип без знака size_type. Когда мы прибавляем отрицательные числа к значению беззнакового типа, ничего хорошего ждать не приходится. Но что именно произойдет? И самое главное – является ли это поведение неопределенным?

Два сценария вызывают озабоченность.

- *Преобразование значения одного целого типа в другой целый тип, если значение выходит за пределы диапазона нового типа*. Это не так опасно, как кажется, поскольку операция приводит не к переполнению, а к оборачиванию (переходу

через одну из границ диапазона). Это верно как для целых без знака, так и – начиная с C++20 – для целых со знаком. Например, 8-разрядное целое со знаком, скажем `int8_t`, может представлять значения от –128 до 127. Если я попытаюсь присвоить ему значение 128, слишком большое, то в результате оборачивания получу –128. Когда значение оборачивается, мы неизбежно получим неверный результат, но хотя бы не неопределенное поведение.

- *Если при выполнении арифметической операции результат выходит за пределы диапазона типа результата.* Для целых без знака мы снова получаем оборачивание. Но для целых со знаком имеет место переполнение, что считается неопределенным поведением.

Поскольку мы прибавляем отрицательные числа `-2` и `-3` к целому без знака `acc`, следует ожидать как минимум какого-то оборачивания, но можно ли ожидать переполнения и, значит, неопределенного поведения? Зависит от того, выходит ли сумма за пределы диапазона типа результата. Давайте внимательнее взглянем на то, что происходит внутри `std::accumulate`, и разберемся, возможно ли такое.

Алгоритм начинается с инициализации аккумулятора `acc` значением размера `2`. `acc` имеет тип `T`, который, как мы вывели ранее, является неспецифицированным целым типом без знака `size_type`. Затем мы прибавляем к `acc` целые со знаком `-2` и `-3`, по одному за раз. Наконец, мы возвращаем `acc`. Ниже эти шаги выписаны:

```
size_type acc = 2; // size_type – беззнаковый тип
acc = acc + -2;    // Сложение числа без знака и отрицательного числа со знаком
acc = acc + -3;    // Сложение числа без знака и отрицательного числа со знаком
return acc;
```

Ну и что происходит, когда мы прибавляем два целых со знаком `-2` и `-3` к значению неизвестного целого беззнакового типа `size_type`? В задаче 14 об аристотелевой сумме частей мы узнали об обычных арифметических преобразованиях и расширении целых типов. Это имеет место и здесь.

Поскольку мы не знаем ширину типов `size_type` и `int`, необходимо рассмотреть три случая.

SIZE_TYPE ПОМЕЩАЕТСЯ В INT

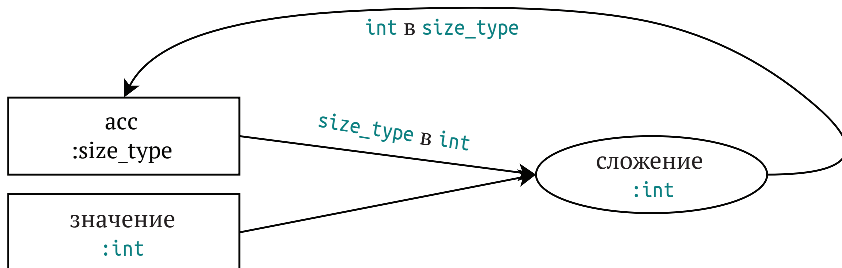
Если `int` может представить все значения типа `size_type`, то `acc` расширяется в `int`, и сложение производится с типом `int`. Поскольку тип `size_type` беззнаковый, это может случиться, только если `size_type` уже, чем `int`. Крайне маловероятно, что реализация выберет типы так, что `size_type` будет уже `int`, но стандарт этого не запрещает. Вот что произошло бы в таком случае.

Сначала мы прибавляем `-2` к `acc`. После расширения `acc` с `size_type` до `int` операнды будут иметь одинаковый тип (`int`), и дальнейших преобразований не потребуется. Складывая `2` и `-2`, мы получаем `0`, который помещается в тип результата `int`. Это значение `int 0` преобразуется обратно в тип `size_type` и сохраняется в `acc`.

Далее мы прибавляем `-3` к `acc`. После расширения `acc` с `size_type` до `int` мы складываем `0` и `-3` и получаем значение `-3`, которое также помещается в тип результата `int`. Значение `int -3` преобразуется обратно в тип `size_type`. Поскольку тип `size_type` беззнаковый, `-3` оборачивается, в результате чего получается `std::numeric_limits<size_type>::max() - 2`, которое сохраняется в `acc` в качестве окончательного результата и возвращается.

Если бы `size_type` совпадал, к примеру, с `uint16_t`, то результат был бы равен `65533`. Это привело бы к оборачиванию и неверному результату, но не к неопределенному поведению.

На диаграмме ниже показаны типы и преобразования, участвующие в этом процессе. Тип `acc` расширяется до `int`, затем к `acc` прибавляется первое значение, и результат преобразуется обратно в тип `size_type`, чтобы можно было сохранить его в `acc`. Затем мы проделываем все это еще раз для второго значения.



Расширение `size_type` в `int` не вызывает проблем, потому что `size_type` помещается в `int`. Сложение целых со знаком может привести к переполнению и неопределенному поведению, если

результат не помещается в `int` (но при тех данных, что фигурируют в этой задаче, такого не происходит). Преобразование `int` в `size_type` может привести к оборачиванию, если результат отрицательный (как в этой задаче) или значение слишком велико (чего в этой задаче не наблюдается).

SIZE_TYPE НЕ ПОМЕЩАЕТСЯ В INT, НО ПОМЕЩАЕТСЯ В UNSIGNED INT

Если тип `int` не может представить все значения типа `size_type`, но `unsigned int` может, то `acc` расширяется в `unsigned int`. Это может произойти, только если `size_type` имеет такую же ширину, как `int` и `unsigned int`. Если бы `size_type` был у́же, то он поместился бы в `int`, а если бы он был шире, то не поместился бы в `unsigned int`.

После того как `acc` был расширен с `size_type` до `unsigned int`, операнды по-прежнему имеют разные типы, поэтому применяются обычные арифметические преобразования. Так как `size_type` и `int` имеют одинаковую ширину, то и ранг у них одинаковый, и `int` преобразуется в `unsigned int` (детали см. в разделе стандарта «Обычные арифметические преобразования»[1]). Складываются значения типа `unsigned int`.

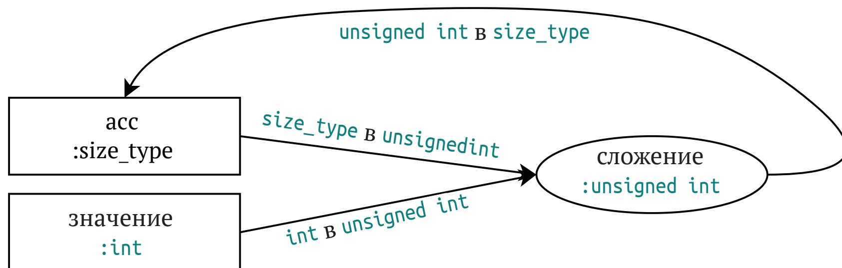
Сначала мы прибавляем `-2` к `acc`. `-2` преобразуется в `unsigned int`, а так как оно отрицательно, то происходит оборачивание в `std::numeric_limits<unsigned int>::max() - 1`. При сложении с `acc`, равным `2`, снова имеет место оборачивание с результатом 0.

Затем мы прибавляем `-3` к `acc`. `-3` преобразуется в `unsigned int`, и, поскольку оно отрицательно, снова происходит оборачивание в `std::numeric_limits<unsigned int>::max() - 2`. При сложении с `acc`, равным `0`, получается значение `std::numeric_limits<unsigned int>::max() - 2`, которое сохраняется `acc` в качестве окончательного результата. В типичной системе с 32-разрядным типом `int` получился бы результат `4294967293`. Мы наблюдали несколько оборачиваний и неверный результат, но неопределенного поведения не было.

Ниже показана диаграмма типов и преобразований, участвующих в этом процессе. `acc` расширяется в `unsigned int`, и первое значение преобразуется в `unsigned int`, они складываются, результат преобразуется обратно в `size_type`, чтобы его можно было сохранить в `acc`. Затем то же самое проделывается для второго значения.

---

[1] *https://timsong-cpp.github.io/cppwp/std20/expr.arith.conv*.



Расширение типа size_type до unsigned int не вызывает проблем, потому что типы size_type и unsigned int оба беззнаковые и одинаковой ширины. Преобразование int в unsigned int может привести к оборачиванию, если значение отрицательно (что и происходит в данной задаче). Сложение чисел без знака может привести к оборачиванию, если результат слишком велик (чего в этой задаче не наблюдается). Преобразование unsigned int в size_type не вызывает проблем, потому что типы unsigned int и size_type оба беззнаковые и одинаковой ширины.

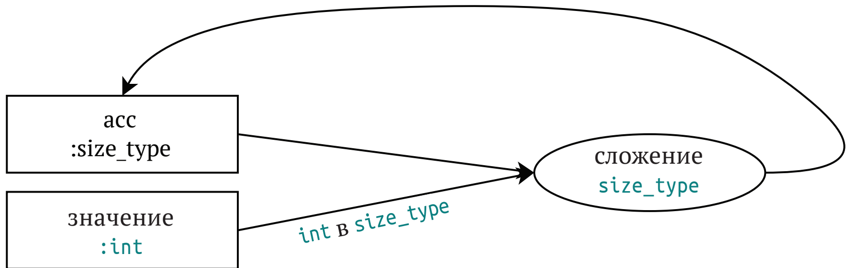
SIZE_TYPE НЕ ПОМЕЩАЕТСЯ НИ В INT, НИ В UNSIGNED INT

Это самый распространенный случай, если, например, ширина int равна 32 разрядам, а size_type – 64-разрядный беззнаковый тип. В этом случае целочисленное расширение size_type не производится, а беспокоиться нужно только об обычных арифметических операциях. Поскольку size_type шире int, то size_type имеет больший ранг, а значит int преобразуется в беззнаковый тип size_type, и складываются значения типа size_type.

Сначала мы прибавляем -2 к acc. -2 преобразуется в size_type, и происходит оборачивание в std::numeric_limits<size_type>::max() - 1. При сложении с acc, равным 2, снова имеет место оборачивание с результатом 0.

Затем мы прибавляем -3 к acc. -3 преобразуется в size_type, и происходит оборачивание в std::numeric_limits<size_type>::max() - 2. При сложении с acc, равным 0, получается значение std::numeric_limits<size_type>::max() - 2, которое сохраняется в acc в качестве окончательного результата. В типичной системе с 32-разрядным типом int и 64-разрядным size_type получился бы результат 18446744073709551613. И снова мы наблюдали несколько оборачиваний и неверный результат, но неопределенного поведения не было.

Ниже показана диаграмма типов и преобразований, участвующих в этом процессе. Первое значение преобразуется в `size_type`, прибавляется к `acc`, и результат сохраняется в `acc`. Затем то же самое проделывается для второго значения.



Преобразование `int` в `size_type` может привести к оборачиванию, если значение отрицательно, что и происходит в данной задаче. Сложение чисел без знака может привести к оборачиванию, если результат слишком велик (чего в этой задаче не наблюдается).

Вы еще не устали?

В предыдущем изложении я опустил много деталей, в частности точные правила, приводящие именно к таким последовательностям преобразований, и все равно объяснение получилось долгим. Если вы следили за ним до самого конца, отлично – значит суть вы уловили. Если в какой-то момент вы плюнули на это дело, тоже ничего страшного – все равно суть до вас дошла! А суть заключается в том, что рассуждать о неявных преобразованиях между целыми трудно, и лучше бы в производственном коде держаться от этого подальше.

Что нам делать?

К сожалению, ни один из основных компиляторов не предупреждает об этом, даже с флагами `-Wall` и `-Wextra` для GCC и Clang и с флагом `/Wall` для MSVC. Некоторые компиляторы, в частности GCC и Clang, имеют дополнительные параметры, например `-Wsign-conversion`, которые не включаются в `-Wall` или `-Wextra` и должны задаваться отдельно. Но, даже задав их, вы все равно не получите никаких предупреждений для этой конкретной задачи. (Но получили бы, если бы шаблон функции `accumulate` был определен в вашем коде, а не в стандартной библиотеке.)

Clang-tidy также не предупреждает об этом, а контролер неопределенного поведения сработал бы, только если бы во время выполнения программы действительно имело место неопределенное поведение – в данном случае этого нет.

Итак, в этой конкретной задаче помощи от инструментов немного, просто нужно знать о потенциальных проблемах при смешении целых типов и знаковости. Тем не менее приведенные ниже советы могут помочь в подобных случаях.

Рекомендации

- Не смешивайте типы со знаком и без знака в арифметических операциях, присваиваниях и сравнениях.
- В дополнение к флагам `-Wall` и `-Wextra` вручную задавайте флаги типа `-Wsign-conversion`.
- Используйте `clang-tidy` и контролеры.
- Помните, что предупреждения и инструменты не всегда показывают все потенциальные проблемы.

Для дополнительного чтения

Обычная путаница с арифметикой

https://shafik.github.io/c++/2021/12/30/usual_arithmetic_confusions.html.

Основные принципы C++ ES.100: не смешивайте знаковые и беззнаковые арифметические операции

https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-mix.

Задача 20

Монстры на марше

monsters-on-the-move.cpp

```cpp
#include <iostream>
struct Monster
{
    Monster() = default;
    Monster(const Monster &other)
    {
        std::cout << "Monster copied\n";
    }
    Monster(Monster &&other)
    {
        std::cout << "Monster moved\n";
    }
};
struct Jormungandr : public Monster
{
    Jormungandr() = default;
    Jormungandr(const Jormungandr &other) : Monster(other)
    {
        std::cout << "Jormungandr copied\n";
    }
    Jormungandr(Jormungandr &&other) : Monster(other)
    {
        std::cout << "Jormungandr moved\n";
    }
};
int main()
{
    Jormungandr jormungandr1;
    Jormungandr jormungandr2{std::move(jormungandr1)};
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

Программа печатает следующий результат:

```
Monster copied
Jormungandr moved
```

Обсуждение

При инициализации второго `Jormungandr` с помощью перемещенного объекта вызывается перемещающий конструктор `Jormungandr`, но копирующий конструктор `Monster`. Почему?

В задаче 8 мы узнали, что при обертывании выражения функцией `std::move` оно превращается в r-значение. Но есть важная деталь: категории значений относятся к выражениям, а не к объектам. Поэтому с самим объектом `jormungandr1` ничего не происходит; только *выражение* `jormungandr1` является l-значением, но выражение `std::move(jormungandr1)` является r-значением. На мой взгляд, жизнь стала бы проще, если бы мы говорили «l-выражение» и «r-выражение», а не «l-значение» и «r-значение».

Когда мы инициализируем `jormungandr2` копированием с помощью выражения `Jormungandr jormungandr2{std::move(jormungandr1)}`, процедура разрешения перегрузки предпочитает перемещающий конструктор `Jormungandr` копирующему, потому что выражение `std::move(jormungandr1)` является r-значением, которое можно переместить. И да, никакой опечатки нет – эта форма всегда называется *инициализацией копированием*, пусть даже в реальности вызван перемещающий конструктор. Нет такого понятия – «инициализация перемещением».

Теперь обратимся к реализации перемещающего конструктора `Jormungandr`. Вы, наверное, уже заметили ошибку: мы забыли обернуть `other` функцией `std::move`:

examples/which-constructors/which-constructors.cpp

```
Jormungandr(Jormungandr &&other) : Monster(std::move(other))
```

Параметр `other` – ссылка на r-значение. Но *выражение* `other`, которое мы передаем конструктору `Monster`, является l-значением! А раз так, то мы не можем использовать перемещающий конструктор, и процедура разрешения перегрузки выбирает вместо него перемещающий конструктор. Однако после обертывания функцией `std::move` выражение `std::move(other)` становится r-значением, и выбирается перемещающий конструктор.

Тогда был бы напечатан такой результат:

```
Monster moved
Jormungandr moved
```

Дополнительный вопрос: что было бы, если структура `Monster` была определена следующим образом?

examples/which-constructors/which-constructors.cpp

```cpp
struct Monster
{
    Monster() = default;
    Monster(const Monster &other) {
        std::cout << "Monster copied\n";
    }
};
```

Поскольку мы явно объявили копирующий конструктор, то не получаем неявно ни конструктора по умолчанию, ни перемещающего конструктора. Мы явно объявили конструктор по умолчанию, но перемещающего конструктора не объявили, а значит, в `Monster` его и не будет.

Когда перемещающий конструктор `Jormungandr` выполняет код `: Monster(std::move(other))`, выражение `std::move(other)` все еще является r-значением. Точнее, как мы видели в задаче 9 о подсчете копий, при оборачивании l-значения функцией `std::move()` оно превращается в разновидность r-значения, которая называется x-значением. Это значит, что объект по-прежнему обладает идентичностью, как l-значение, но при этом его содержимое разрешено перемещать, как для r-значения. Если бы в структуре `Monster` был перемещающий конструктор (с параметром типа `Monster &&`), то процедура разрешения перегрузки предпочла бы его копирующему конструктору (с параметром типа `const Monster &`), когда передано r-значение. Но, поскольку перемещающего конструктора нет, а константным ссылкам на l-значение разрешено связываться с r-значениями, используется копирующий конструктор.

И мы снова возвращаемся к результату:

```
Monster copied
Jormungandr moved
```

Для дополнительного чтения

Lvalues, Rvalues, Glvalues, Prvalues, Xvalues, Help!
https://blog.knatten.org/2018/03/09/lvalues-rvalues-glvalues-prvalues-xvalues-help/.

Различие между отсутствием перемещающего конструктора и удаленным перемещающим конструктором
https://blog.knatten.org/2021/10/15/the-difference-between-no-move-constructor-and-a-deletedmove-constructor/.

Задача 21

Измерение некоторых символов

sizing-up-some-characters.cpp

```
#include <iostream>

void serialize(char characters[])
{
    std::cout << sizeof(characters) << "\n";
}
int main()
{
    char characters[] = {'a', 'b', 'c'};
    std::cout << sizeof(characters) << "\n";
    std::cout << sizeof(characters) / sizeof(characters[0]) << "\n";
    serialize(characters);
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

Поведение программы зависит от реализации! Но на машине с 64-разрядным процессором она, скорее всего, напечатает следующий результат:

```
3
3
8
```

Обсуждение

В программе определен массив `characters` с тремя элементами. Затем мы разными способами печатаем размер этого массива. Что оператор `sizeof` делает в каждом случае? И что будет напечатано на вашем компьютере?

Прежде всего определяется массив `char characters[] = {'a', 'b', 'c'}`. Затем мы печатаем размер этого массива, пользуясь оператором `sizeof`. Оператор `sizeof` возвращает количество байтов, занятых его операндом, в данном случае – `3`, поскольку каждый символ `char` занимает один байт, а всего их три.

Но погодите – в задаче 15 об аристотелевой сумме частей мы узнали, что стандарт не определяет размеры фундаментальных типов, и как же тогда мы можем быть уверены, что ответ равен `3`? Ведь ширина `char` вполне могла бы быть равна 16 разрядам. Но, как выясняется, `sizeof(char)` по определению всегда возвращает `1`. Поэтому ширина «байта», о котором толкует стандарт C++, составляет не восемь бит, а столько, сколько занимает `char`.

Далее мы печатаем `sizeof(characters) / sizeof(characters[0])`. Это старинная идиома, применяемая для подсчета элементов в массиве C или C++, и в этой задаче гарантированно печатается `3`. Оно и понятно: `sizeof(characters)` возвращает `3` (как мы только что видели), а `sizeof(characters[0])` эквивалентно `sizeof(char)`, которое всегда равно `1`. (Начиная с C++17, мы можем подсчитывать элементы в массиве также с помощью нового шаблона функции `std::size`, который немного упрощает код: `std::cout << std::size(characters)`.)

Наконец, мы хотим сериализовать наши символы, поэтому передаем их функции `void serialize(char characters[])`. Эта функция хочет знать размер массива для целей сериализации и пользуется оператором `sizeof(characters)`, как и в `main`. Но почему теперь печатается `8` (на моем компьютере), а не `3`? Потому что тип параметра `char characters[]` на самом деле не «массив символов», а «указатель

на `char`»! В стандарте имеется следующий абзац, касающийся типов параметров функции:

> После определения типа каждого параметра все параметры типа «массив T» или типа функции T заменяются на «указатель на T».

Таким образом, `sizeof(characters)` в функции `serialize` – это размер указателя, а не размер массива, потому что в этот момент тип параметра `characters` – «указатель на `char`». Это зависящая от реализации часть задачи, потому что стандарт не определяет размер указателя. В большинстве современных компьютеров ширина указателя равна 64 разрядам, или 8 байтам, так что на вашем компьютере, скорее всего, будет напечатано `8`.

А можем ли мы использовать более мощный, на первый взгляд, шаблон функции `std::size`, чтобы получить истинное число элементов в массиве? Не можем, потому что `characters` – указатель, а не массив, а `std::size` определен только для коллекций и представлений, имеющих функцию-член `.size()`, и для массивов. Вызов `std::size(characters)` просто не откомпилируется, что, конечно, лучше, чем возврат неверного результата.

Рекомендации

- Массивы в C++ крайне ограничены, и при передаче их функциям информация о размере теряется. Предпочитайте им такие типы, как `std::vector` или `std::array`.
- Включайте в проекте предупреждения. GCC и Clang предупредят об использовании `sizeof(characters)` в `serialize`. Вот что говорит GCC: «warning: sizeof on array function parameter will return size of 'char *' instead of 'char[]'». (На момент написания книги я не смог заставить MSVC дать аналогичное предупреждение.).

Для дополнительного чтения

Оператор sizeof

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

Шаблон функции std::size

https://en.cppreference.com/w/cpp/iterator/size.

Задача 22

Космический корабль-призрак

the-phantom-spaceship.cpp

```cpp
#include <iostream>
#include <string>

struct GameObject
{
    GameObject() { std::cout << "Created a " << getType() << "\n"; }
    void render() const { std::cout << "Rendered a " << getType() << "\n"; }
    virtual std::string getType() const { return "GameObject"; }
};
class Spaceship : public GameObject
{
    std::string getType() const override { return "Spaceship"; }
};
void display(const GameObject &gameObject) { gameObject.render(); }

int main()
{
    GameObject gameObject;
    Spaceship spaceship;
    display(gameObject);
    display(spaceship);
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

Программа напечатает следующий результат:

```
Created a GameObject
Created a GameObject
Rendered a GameObject
Rendered a Spaceship
```

Обсуждение

В этой задаче предполагается, что вы попытались включить в свою игру протоколирование, чтобы отслеживать создание и отрисовку объектов игры. Вы решили использовать паттерн проектирования Шаблонный метод – отличный способ добиться полиморфного поведения. В этом паттерне базовый класс определяет общее поведение в конкретной функции, а та вызывает восполняющие детали виртуальные функции, которые можно переопределить в производных классах. Например, в коде выше функция `render` – шаблонный метод, который вызывает виртуальную функцию `getType`, определяющую детали в конкретном производном типе.

Это прекрасно работает в функции `render`, но в конструкторе что-то явно не так: он печатает `Created a GameObject` как при создании `GameObject`, так и при создании `Spaceship`. Что именно происходит и может ли это быть неопределенным поведением?

В `main` при инициализации объекта типа `Spaceship` сначала инициализируется часть объекта, относящаяся к базовому классу `GameObject`, а затем часть, относящаяся к классу `Spaceship`. Во время инициализации части `GameObject` части, относящейся к `Spaceship`, еще не существует. С точки зрения C++ просто создается объект типа `GameObject`.

Так что же происходит при вызове виртуальной функции-члена `getType()` в конструкторе `Spaceship`, который при обычных обстоятельствах был бы передан классу `Spaceship`, но того пока даже не существует? Неопределенное поведение? К счастью, нет – это поведение точно определено. Когда виртуальная функция вызывается из конструктора или деструктора, происходит обращение к функции, определенной в данном классе, а не к той, что переопределена в производном классе. Поэтому в конструкторе `GameObject` функция `GameObject::getType()` вызывается и при создании `GameObject`, и при создании `Spaceship`. В обоих случаях печатается `Created a GameObject`.

Однако в функции `render` мы имеем дело с полностью инициализированными объектами, и виртуальная функция работает как и положено. При отрисовке `GameObject` печатается сообщение `Rendered a GameObject`, а при отрисовке `Spaceship` – сообщение `Rendered a Spaceship`.

Как исправить ситуацию? Прежде чем говорить о способах, позволяющих сделать фактический тип создаваемого объекта доступным конструктору `GameObject`, стоит задаться вопросом, а следует ли вообще разрешать создание объектов типа `GameObject`? Имеет ли в игре смысл создавать «абстрактные» игровые объекты? Или все-таки только конкретные объекты: космический корабль, астероид, космонавта и т. д.? Если мы сделаем `GameObject::getType()` *чисто виртуальной функцией* и не будем давать ей определение, то `GameObject` станет абстрактным классом, экземпляр которого невозможно создать:

examples/spaceship/spaceship-pure.cpp

```
virtual std::string getType() const = 0;
```

Но как превращение `GameObject::getType()` в чисто виртуальную функцию повлияет на эту программу? Попробовав это сделать, вы увидите, что создание `GameObject gameObject` перестало компилироваться. Нельзя создавать объекты абстрактных классов – только конкретных производных классов, таких как `Spaceship`. Кроме того, создание объекта `Spaceship spaceship` теперь приводит к неопределенному поведению, потому что конструктор базового класса `GameObject` теперь вызывает чисто виртуальную функцию. На практике вы, скорее всего, увидите либо предупреждение компилятора о вызове чисто виртуальной функции, либо ошибку компоновщика из-за отсутствия определения `GameObject::getType`, либо ошибку времени выполнения типа «Вызвана чисто виртуальная функция». Но, как всегда в случае неопределенного поведения, конкретика зависит от платформы и компилятора. Но является ли переход от неверного результата к неопределенному поведению благом? Я думаю, что в данном случае да, поскольку при правильной настройке конвейера непрерывной интеграции вы, скорее всего, обнаружите и исправите ошибку.

А как ее правильно исправить? Поскольку сам конструктор `GameObject` никак не может узнать, какой производный тип фактически конструируется, мы должны как-то передать ему эту информацию. Проще всего прямо передать ее конструктору с помощью параметра, например:

examples/spaceship/spaceship-parameter.cpp

```
protected:
    GameObject(std::string_view type) {
        std::cout << "Created a " << type << "\n";
}
```

Этот новый конструктор защищен, потому что предназначен только для вызова из производных классов. В классе `Spaceship` это делается так:

examples/spaceship/spaceship-parameter.cpp

```
public:
    Spaceship() : GameObject("Spaceship"){}
```

(Заметим, что строка «Spaceship» теперь встречается дважды: в конструкторе и в функции `getType`, поэтому стоило бы сделать ее членом класса `Spaceship` с модификаторами `static constexpr`.)

Если мы все-таки хотим создавать «голые» объекты `GameObject`, то должны вернуть конструктор по умолчанию:

examples/spaceship/spaceship-parameter.cpp

```
public:
    GameObject() : GameObject("GameObject"){}
```

Но при этом возникает новая проблема: если вы забудете добавить пользовательский конструктор в `Spaceship`, то конструктор `Spaceship` по умолчанию просто воспользуется конструктором `GameObject` по умолчанию, и мы снова увидим сообщение `GameObject created` при создании объекта `SpaceShip` – еще одна причина не разрешать независимое конструирование `GameObject`.

Наконец, если проблема заключается в том, как использовать информацию из производного класса в базовом классе, не прибегая к динамическому полиморфизму, то никакое обсуждение не может пройти мимо идиомы курьезного рекуррентного шаблона (Curiously Recurring Template Pattern – CRTP). У нас здесь нет места для ее подробного обсуждения, но идея состоит в том, чтобы сделать базовый класс шаблоном, а затем использовать производный класс в качестве параметра шаблона в своем собственном базовом классе. Теперь базовый класс может узнать – и даже статически, – объект какого производного класса создается, просто взглянув на параметр шаблона! Вот полный пример:

examples/spaceship/spaceship-crtp.cpp

```
#include <iostream>

struct GameObject {
    virtual void render() const = 0;
};
template<typename Derived>
struct LoggingGameObject : public GameObject {
    LoggingGameObject() {
        std::cout << "Created a " << Derived::typeName << "\n";
    }
    void render() const override {
        std::cout << "Rendered a " << Derived::typeName << "\n";
    }
};
struct Spaceship : public LoggingGameObject<Spaceship> {
    static constexpr auto typeName = "Spaceship";
};
void display(const GameObject &gameObject) { gameObject.render(); }

int main() {
    Spaceship spaceship;
    display(spaceship);
}
```

Более подробное обсуждение CRTP смотрите в «Википедии»[1].

Рекомендации

- Включайте предупреждения. Например, компиляторы GCC и Clang умеют предупреждать о вызовах чисто виртуальных функций в конструкторах и деструкторах.
- Проверьте несколько компиляторов и платформ в конвейере непрерывной интеграции. Разные компиляторы и компоновщики выдают различные предупреждения и ошибки.

[1] https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern.

Для дополнительного чтения

Паттерн проектирования Шаблонный метод
https://en.wikipedia.org/wiki/Template_method_pattern.

Курьезный рекуррентный шаблон (CRTP)
https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern.

Задача 23

Доброе начало полдела откачало

off-to-a-good-start.cpp

```cpp
#include <iostream>

struct Logger {};
struct Configuration {};
Logger initializeLogger()
{
    std::cout << "Initializing logger\n";
    return Logger{};
}
Configuration readConfiguration()
{
    std::cout << "Reading configuration\n";
    return Configuration{};
}
void startProgram(Logger logger, Configuration configuration)
{
    std::cout << "Starting program\n";
}
int main()
{
    startProgram(initializeLogger(), readConfiguration());
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

У этой программы неспецифицированное поведение! Но напечатать она может одно из двух. Первый вариант:

```
Initializing logger
Reading configuration
Starting program
```

Второй вариант:

```
Reading configuration
Initializing logger
Starting program
```

Обсуждение

В этой задаче мы хотим запустить программу, которая ожидает, что регистратор уже инициализирован и конфигурация прочитана. Если то и другое сделано до начала работы программы, все отлично. Порядок, в котором настраивается регистратор и читается конфигурация, здесь, похоже, не имеет значения; но что, если бы имел? Например, что, если в процессе инициализации регистратора создается его глобальный экземпляр, который мы используем при чтении конфигурации? Можно ли рассчитывать на то, что две функции будут вызываться в порядке, указанном в программе? К сожалению, нет, потому что порядок вычисления аргументов функции не специфицирован. И это не чистая теория; при тестировании в Linux GCC вычислял аргументы справа налево, с Clang – слева направо. Онлайновую демонстрацию можете посмотреть на сайте Compiler Explorer[1].

Но хаос все-таки царит не повсеместно, кое-какие гарантии нам дают.

1. Гарантируется, что инициализация параметров со всеми побочными эффектами завершается до начала выполнения функции. А следовательно, `initializeLogger` и `readConfiguration` будут выполнены к моменту, когда начнет выполняться `startProgram`, и мы, например, можем не опасаться, что на печати сообщение `Initializing logger` будет чередоваться со `Starting program`. Кажется очевидным, но ведь это же C++, так что нужно быть начеку.

[1] https://www.godbolt.org/z/P43cETdWd.

2. Гарантируется, что инициализация одного параметра со всеми побочными эффектами завершается до начала инициализации другого параметра. То есть либо `initializeLogger` полностью завершится до начала `readConfiguration`, либо наоборот.
3. Здесь нет неопределенного поведения, просто выражения аргументов могут вычисляться в любом порядке.

Мы знаем, что `initializeLogger` и `readConfiguration` завершаются до начала `startProgram`, но не знаем, в каком порядке. Не гарантируется даже, что порядок будет одинаковым при каждом запуске программы, хотя ни компилятор, ни компьютер не менялись! Маловероятно, что компилятор будет генерировать код, не детерминированный в этом отношении, но стандарт этого не запрещает. Возможно также, что порядок изменится с выходом новой версии компилятора, если автор, к примеру, решит, что вычислять параметры справа налево эффективнее, чем слева направо.

Что тут можно сделать? Если порядок действительно важен, то можете просто вызывать функции одну за другой в том порядке, в каком хотите:

examples/deterministic-order/deterministic-order.cpp

```
int main()
{
    const auto logger = initializeLogger();
    const auto configuration = readConfiguration();
    startProgram(logger, configuration);
}
```

Заметим, что компилятор не выдает никаких предупреждений по этому поводу. Не обладая волшебной способностью узнать семантику программы в целом, компилятор не может сказать, имеет ли порядок вычисления параметров функции какое-то значение для вашей программы.

Рекомендации

- Не полагайтесь на определенный порядок вычисления выражений, переданных в качестве аргументов функции.
- Не предполагайте, что выражения вычисляются в каком-то определенном порядке только потому, что так было в ваших тестах.

Для дополнительного чтения

Порядок вычисления
https://en.cppreference.com/w/cpp/language/eval_order.

Задача 24

Специальная теория струн

a-specialized-string-theory.cpp

```
#include <iostream>

template<typename T>
void serialize(T&) { std::cout << "template\n"; } // 1
template<>
void serialize<>(const std::string&) {
    std::cout << "specialization\n";
} // 2
void serialize(const std::string&) {
    std::cout << "normal function\n";
} // 3
int main()
{
    std::string hello_world{"Hello, world!"};
    serialize(hello_world);
    serialize(std::string{"Good bye, world!"});
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

Программа напечатает следующий результат:

```
template
normal function
```

Обсуждение

Возможно, вы еще не забыли нашу библиотеку сериализации из задачи 2 «Теория струн». А тем временем явился эксперт по C++ и решил, что с шаблонами все будет гораздо лучше. Так это или не так – другой вопрос, но уж разрешение перегрузки точно станет интереснее!

В этой задаче мы видим три определения `serialize`. Первое (с меткой «1») – основной шаблон функции `serialize`. Следующее (с меткой «2») – явная специализация с пустым тегом `template<>`. Наконец, меткой «3» обозначена нешаблонная функция `serialize`.

Так как имя `serialize` перегружено, при обращении к этому имени процедура разрешения перегрузки должна решить, что именно вызвать. Как мы знаем из задачи 2, для этого она создает множество функций-кандидатов с этим именем, а затем выбирает лучшую подходящую функцию из этого множества. Ясно, что обычная, нешаблонная функция `serialize` (3) является кандидатом, но как обстоит дело с шаблоном функции? И как компилятор может отличить специализацию шаблона функции от обычной функции – ведь сигнатуры обеих одинаковы?

Когда в разрешении перегрузки участвует шаблон функции, компилятор рассматривает все *первичные шаблоны* с таким именем. В этот момент он не рассматривает специализации, так что рассматривается только первичный шаблон (1), но не специализация (2). Если бы существовал другой перегруженный вариант первичного шаблона, то он бы тоже рассматривался, но в этой задаче есть только один. А мог бы, например, существовать первичный шаблон с указателями:

```
template<typename T>
void serialize(T*) { std::cout << "pointer\n"; }
```

То, что это именно первичный шаблон, мы можем определить по непустым угловым скобкам в части `template<typename T>`, как и поступили в отношении первичного шаблона в этой задаче.

Поскольку первичный шаблон (1) – единственный первичный шаблон в этой задаче, мы нашли шаблон функции `serialize`, кото-

рый войдет в множество кандидатов для разрешения перегрузки. Но чтобы двигаться дальше, нужна конкретная сигнатура функции с фактическими типами, а не только шаблон, принимающий произвольный тип T. Поэтому далее производится вывод типа для serialize с целью узнать, как будет выглядеть потенциальная конкретизация. Это называется синтезом объявления. Это синтезированное объявление добавляется в множество функций-кандидатов (вместе с нешаблонной функцией serialize), и регулярная процедура разрешения перегрузки продолжается. Посмотрим, как работает вывод типа и разрешение перегрузки для каждого из двух вызовов serialize.

Сначала взглянем на вызов, в котором передается объект:

```
serialize(hello_world);
```

Выражение hello_world – это l-значение типа std::string. Как мы узнали в задаче 13 «Постоянная борьба», результатом вывода T является std::string, а синтезированное объявление имеет сигнатуру void serialize(std::string&). Теперь у нас есть два кандидата на разрешение перегрузки:

```
void serialize(std::string&); // синтезировано из шаблона (1)
void serialize(const std::string&); // нешаблонная функция (3)
```

Поскольку hello_world не константа, разрешение перегрузки предпочитает шаблонную версию serialize, которая принимает неконстантную ссылку.

Теперь, когда процедура разрешения перегрузки решила использовать шаблон функции, настало время определить специализацию. Следует ли конкретизировать и использовать первичный шаблон функции (1) или явную специализацию (2)? В данном случае выбор прост, так как нам нужна специализация с сигнатурой void serialize(std::string&), но наша явная специализация имеет сигнатуру void serialize(const std::string&). Мы вынуждены использовать первичный шаблон функции и неявную конкретизацию с T = std::string, что дает void serialize(std::string&). При вызове этой конкретизации печатается template.

Далее рассмотрим вызов, в котором передается временная строка:

```
serialize(std::string{"Good bye, world!"});
```

Выражение std::string{"Good bye, world!"} – это pr-значение типа std::string, поэтому результатом вывода T является std::string, как

и в предыдущем вызове. Мы снова имеем те же два кандидата на разрешение перегрузки:

```
void serialize(std::string&); // синтезировано из шаблона (1)
void serialize(const std::string&); // нешаблонная функция (3)
```

Поскольку `std::string{"Good bye, world!"}` – pr-значение, неконстантную ссылку в шаблоне функции нельзя связать с ним. А константную ссылку в обычной функции можно, поэтому процедура разрешения перегрузки ее и выбирает. Так как был выбран не шаблон, конкретизация не производится, и печатается сообщение `normal function`.

Дополнительный вопрос: что было бы, если бы шаблон функции принимал также константную ссылку?

examples/template-vs-function/template-vs-function.cpp

```
template<typename T>
void serialize(const T&) { std::cout << "template\n"; } // 1
```

Тогда синтезированное объявление имело бы сигнатуру `void serialize(const std::string&)`, в точности такую, как у нешаблонной функции, и процедуре разрешения перегрузки пришлось бы выбирать между ними:

```
void serialize(const std::string&); // синтезировано из шаблона (1)
void serialize(const std::string&); // нешаблонная функция (3)
```

Если при разрешении перегрузки обнаруживаются два одинаково хороших подходящих кандидата, то обычно имеет место ошибка компиляции. Так откомпилируется ли вообще эта программа?

На самом деле здесь все нормально. Когда в разрешении перегрузки участвуют шаблоны, имеет место особый случай: если функция F1 не является специализацией шаблона, а F2 является специализацией шаблона, то F1 лучше F2. Для обоих вызовов выбирается нешаблонная функция, и программа печатает

```
normal function
normal function
```

Заметим, что ни в одном из рассмотренных случаев явная специализация шаблона функции даже не рассматривалась – она вообще никогда не рассматривается. Это может противоречить интуиции, поэтому в общем случае лучше использовать нешаблонные пере-

груженные варианты, а не явные специализации. Смотрите первые две ссылки в разделе «Для дополнительного чтения», где приведены дополнительные детали и еще худшие примеры такого рода недоразумений.

Рекомендация

- Предпочитайте нешаблонные перегруженные варианты явным специализациям.

Для дополнительного чтения

Почему не нужно специализировать шаблоны функций?
http://www.gotw.ca/publications/mill17.htm.

Полная специализация шаблонов функций
https://www.modernescpp.com/index.php/full-specialization-of-function-templates/.

Основные принципы T.144: не специализируйте шаблоны функций
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#t144-dont-specialize-functiontemplates.

Задача 25

Слабо типизированная, сильно озадачивающая

weakly-typed-strongly-puzzling.cpp

```
#include <iostream>

int main()
{
    std::cout << +!!"";
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

Программа напечатает следующий результат:

```
1
```

Обсуждение

Что, что? Это вообще допустимый C++? Что означает этот код и почему он печатает `1`?

Хотя C++ – статически типизированный язык, он не особенно сильно типизирован. Как мы видели на протяжении всей книги, C++ весьма охотно прибегает к неявным преобразованиям типов, его даже просить не надо. Например, в задаче 14 об аристотелевой сумме частей мы видели, что сумма двух `char` не `char`, в задаче 17 «Строгий указ» – что следует объявлять конструкторы явными (`explicit`), чтобы они не принимали участия в неявных преобразованиях, а в задаче 19 «Маленькая сумма» – что неявные преобразования арифметических типов могут приводить к неожиданным результатам.

Эту готовность мы унаследовали от тех времен, когда C++ проектировался как расширение C. Если бы дело происходило сейчас, то это изобилие неявных преобразований стояло бы на одном из первых мест в списке вещей, которые мы хотели бы изменить.

Так какие же неявные преобразования вступают в игру здесь, каков окончательный тип этого выражения и почему результат равен 1?

Начнем справа, с выражения `""`. Это строковый литерал, а, как было сказано в задаче 2 «Теория струн», типом строкового литерала является «массив из `n` `const char`», где в данном случае `n` равно `1` (неявный завершающий `\0`).

Далее идет оператор `!`, который вычисляет отрицание своего операнда. Но что такое отрицание массива символов? Это же не имеет смысла, правда ведь? Да, не имеет, но операнд оператора `!` в данном контексте преобразуется в `bool`, и только потом берется его отрицание. Преобразование массива в `bool` тоже не имеет смысла, но, как мы видели в задаче 2, массивы можно неявно преобразовать в указатели, а вот указатель уже допускает преобразование в `bool`. Когда указатель преобразуется в `bool`, результатом является `false`, если это нулевой указатель, и `true` в противном случае. Поскольку в данном случае указатель ненулевой (он указывает на строковый литерал), он преобразуется в `true`. Наконец, оператор `!` вычисляет отрицание, и мы получаем `false`. Можно было бы возразить, что вся эта последовательность преобразований сама по себе не имеет особого смысла, и я с этим соглашусь, но мы, по крайней мере, можем понять, как

работает каждый шаг и каким образом их комбинация дает такой результат.

Далее следует еще один `!`, который вычисляет отрицание нашего `bool false`, что дает `bool true`.

И наконец, унарный `+`. Операнд унарного `+` подвергается целочисленному расширению, которое для `bool` означает преобразование `false` в `0` и `true` в `1`. Сам оператор ничего не делает; его результат совпадает с его операндом (после расширения типа). Таким образом, наше `bool true` превращается в `int 1`, которое и печатается.

Должен ли компилятор предупреждать о чем-то из вышеизложенного – другой вопрос, но ни один из основных компиляторов этого не делает, даже с флагами `-Wall -Wextra` для GCC и Clang и с флагом `/Wall` для MSVC. Возможно, ваш компилятор выдаст какие-то предупреждения, если включить их явно. Например, Clang будет предупреждать о преобразовании строки в `bool`, если задать флаг `-Wstring-conversion` (который не включается ни в `-Wall`, ни в `-Wextra`). Посмотрите в документации, какие предупреждения поддерживает ваш компилятор.

Рекомендации

- Включайте как можно больше предупреждений.
- Помните, что не все предупреждения включаются на максимальном уровне предупреждений компилятора.
- Не пишите такой код в реальных программах!

Для дополнительного чтения

CppReference о неявных преобразованиях

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

Последовательности неявных преобразований в стандарте

https://timsong-cpp.github.io/cppwp/std20/over.best.ics.

Предметный указатель

Symbols

.bss секция 31
-Wall флаг 28, 39, 103
-Wextra, флаг 28, 103
-Wpedantic флаг 28
-Wreorder-ctor флаг 39
-Wreorder флаг 39
-Wsign-conversion флаг 103
-Wstring-conversion флаг 131
/W флаг 28

А

автоматическая продолжительность хранения 30, 43
арифметические операции
  не смешивать знаковые и беззнаковые типы 98
  обычные арифметические преобразования 78, 99
  целые со знаком и неопределенное поведение 13
атомарные типы, гонки за данные 82

Б

блокировки 82

В

векторы
  инициализация списком 88
  использование вместо массивов 111
  конструкторы 88
вывод аргумента шаблона 74
выражение с отбрасываемым значением 60

Г

глобальные переменные
  динамическая инициализация 43
  инициализация нулем 30
  статическая продолжительность хранения 30, 42
  уничтожение 44
гонка за данные 82

Д

двоичный интерфейс приложений (ABI) 56
деструктор
  и неполное конструирование 70
  объявление виртуальным 35
  перемещенные объекты 94
  порядок выполнения 38, 44, 66
динамическая инициализация 43

Е

единицы трансляции
  и динамическая инициализация 43

И

инициализация
динамическая 43
константой 30
нулем 30
параметры и порядок вызова функций 120
переменных и неопределенное поведение 26
перемещающие конструкторы 48
порядок 43, 70
по умолчанию 18
прямая инициализация
и неявное преобразование 90
и перемещающие конструкторы 49
списки инициализации в фигурных скобках 86
списки инициализации членов 18
списком и разрешение перегрузки 87
статическая 30, 45
явная 18

К

классы
курьезный рекуррентный шаблон (CRTP) 116
паттерн проектирования Шаблонный метод 114
порядок инициализации членов 70
создание экземпляров и абстрактные классы 115
компилятор *см.* разрешение перегрузки, неопределенное поведение, предупреждения
Compiler Explorer 15
двоичный интерфейс приложений (ABI) 56
и неспецифицированное поведение 14
инициализация константой 30
и поведение, зависящее от реализации 14
оптимизации 26
оптимизация возвращаемого значения 55
конструкторы
вектора 88
копирующие 49, 55, 106
материализация временных объектов 64
неполное конструирование и уничтожение 70
перемещающие 48, 95, 106
последовательность определенных пользователем преобразований 21
прямая инициализация и явное объявление 90
разрешение перегрузки 86, 106
списки инициализации членов 18, 38
контролеры 27, 67, 83, 104
курьезный рекуррентный шаблон (CRTP) 116

Л

линтинг 28
локальные переменные
автоматическая продолжительность хранения 30
инициализация 43
область видимости 31, 94
оптимизация именованного возвращаемого значения (NRVO) 55
статическая продолжительность хранения 43
стек 26
уничтожение 44

М

массивы
ограничения в C++ 111
преобразование 21, 130
разрешение перегрузки 21
материализация временного объекта 61
мьютекс 82

Н

неопределенное поведение
гонка за данные 82
деструкторы 35
инициализация переменных 26
непрерывная интеграция
предупреждения 28
неспецифицированное поведение 14
неявное преобразование
объяснение 130
обычные арифметические преобразования 78
разрешение перегрузки 21, 87

О

область видимости переменных 31, 44, 94
объекты
время жизни временных объектов 64
перемещенные 94
уничтожение 34
обычные арифметические преобразования 78, 99
! оператор 130
+ оператор 131
оптимизация, включение в компиляторе 27
оптимизация именованного возвращаемого значения (NRVO) 55
оптимизация коротких строк 95
оптимизация неименованного возвращаемого значения (URVO) 54
отрицание массива 130

П

параметры
инициализация и порядок вызова функций 120
параметры шаблона 74
типы параметров шаблона 74
первичные шаблоны 124
переменные
автоматическая продолжительность хранения 30, 43
динамическая инициализация 43
инициализация и неопределенное поведение 26
инициализация константой 30
инициализация нулем 30
локальные и стек 26
область видимости 31, 44, 94
порядок инициализации 14
присваивание функциям 60
статическая инициализация 30
статическая продолжительность хранения 30, 42
уничтожение 44
перемещающие конструкторы
объяснение 48
перемещенные объекты 94
разрешение перегрузки 106
поведение *см.* неопределенное поведение
зависящее от реализации 14
неспецифицированное 14
подобъекты
и перемещающие конструкторы 49
уничтожение 35

полиморфизм
  курьезный рекуррентный
    шаблон (CRTP) 116
  паттерн проектирования
    Шаблонный метод 114
порядок
  вызова функций 120
  выполнения
    деструкторов 38, 44, 66
  вычисления аргументов
    функции 14
  инициализации 43
  инициализации глобальных
    переменных 14
  предупреждения 39
  списки инициализации членов 38
потоки, гонка за данные 82
предупреждения
  гонки за данные 83
  о sizeof 111
  о вызове чисто виртуальной
    функции 115, 117
  о порядке инициализации
    членов 39
  о преобразовании string
    в bool 131
  о смешении знаковых и
    беззнаковых типов 103
  трактовка как ошибок 28
преобразование *см.*
  неявное преобразование
  инициализация списком и
    разрешение перегрузки 87
  не смешивать знаковые и
    беззнаковые типы 98
  обычные арифметические
    преобразования 78, 99
  последовательность
    определенных пользователем
    преобразований 21
  последовательность
    стандартных
    преобразований 21
  прямая инициализация и явное
    объявление 90
привязка
  и материализация временных
    объектов 64
  ссылок к pr-значениям 64
пропуск копирования 54
  инициализация копированием
    и прямая инициализация 91
  и перемещающие
    конструкторы 106
прямая инициализация
  и неявное преобразование 90
  и перемещающие
    конструкторы 49
прямая инициализация
  иинициализация
    копированием 91

Р

разрешение перегрузки
  инициализация списком 87
  копирующие конструкторы 106
  перемещающие
    конструкторы 106
  прямая инициализация 90
  строки 21, 125

С

сериализация
  символов 110
  строк 20, 125
символы
  обычные арифметические
    преобразования 78
  оператор sizeof 110
  сериализация 110
синтезированное объявление 125
специализация, разрешение
  перегрузки 124
{} списки инициализации
  в фигурных скобках 86
списки инициализации
  в фигурных скобках 86

список инициализаторов 87
статическая инициализация 30, 45
статическая продолжительность хранения 30, 42
стек
  локальные переменные 26
  указатели и пропуск копирования 56
Страуструп Бьярн 9
строгая типизация 90
строки
  перемещенные объекты 95
  разрешение перегрузки 21, 125
  сериализация 20, 125
  строковые литералы и класс string 21

Т

трактовка предупреждений как ошибок 28

У

указатель
  и оператор sizeof 111
  и перемещающие конструкторы 56, 95
  преобразование в bool 130
  преобразование массива в 21

Ф

Фаллер Бьярн 92
фиаско порядка статической инициализации (SIOF) 45
функции
  виртуальные 34, 114
  выражение с отбрасываемым значением 60
  деструкторы 34
  избегать вызовов вручную 71
  паттерн проектирования Шаблонный метод 114
  подходящие 20
  порядок вызова 120
  порядок вычисления аргументов 14
  присваивание 60
  разрешение перегрузки 20

Ц

целые
  не смешивать знаковые и беззнаковые типы 98
  обычные арифметические преобразования 78, 99
  переполнение арифметики со знаком и неопределенное поведение 12
  размер 14
  целочисленное расширение 78, 99, 131

Ш

шаблоны
  вывод параметров 74
  курьезный рекуррентный шаблон (CRTP) 116
  первичные 124
шаблоны функций
  вывод типа аргумента 74
  разрешение перегрузки 124
явная инициализация 18

A

auto правила вывода 76

C

C5038 параметр предупреждения 39
Clang
  включение предупреждений 28, 111, 117
  двоичный интерфейс приложений (ABI) 56
  предупреждения о sizeof 111
  предупреждения о вызовах чисто виртуальных функций 117
  предупреждения о порядке инициализации членов 39
  предупреждения о

преобразовании string
в bool 131
предупреждения о смешении
знаковых и беззнаковых
типов 103
clang-tidy 28, 45, 104
cppcoreguidelines-interfaces-
global-init 45
cppquiz.org 8
CppReference.com 43
C++ Pub Quiz 8

F

for цикл, ошибка при работе
с диапазонами 67
fstream 71

G

GCC
включение
предупреждений 28, 111, 117
двоичный интерфейс
приложений (ABI) 56
предупреждения о sizeof 111
предупреждения о вызовах
чисто виртуальных
функций 117
предупреждения о порядке
инициализации членов 39
предупреждения о смешении
знаковых и беззнаковых
типов 103

I

is_always_lock_free 83
is_lock_free 83
Itanium C++ ABI 56

L

LLVM 83
l-значения
определение 48
перемещающие
конструкторы 48, 106

M

MemorySanitizer 27
move 48, 94
MSVC
двоичный интерфейс
приложений (ABI) 56
предупреждения 27, 39

P

pr-значения
материализация временного
объекта 61
определение 54
привязка ссылое 61
присваивание переменных
функциям 60

R

RAII (захват ресурса является
инициализацией) 36
r-значения
определение 48

S

sizeof 110
size шаблон функции 111
static 32
strong_type библиотека 92

T

ThreadSanitizer 83

U

unique_ptr 71

V

vector 88

X

x-значения 54, 107

Книги издательства «ДМК Пресс» можно купить оптом
в книготорговой компании «Галактика» (представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
Тел. **+7(499) 782-38-89**.
Электронная почта: **books@alians-kniga.ru**.

Андерс Шау Кнаттен

Хорошо ли вы знаете C++?

Главный редактор *Мовчан Д. А.*
Зам. главного редактора *Яценков В. С.*
editor@dmkpress.com
Перевод *Слинкин А. А.*
Корректор *Абросимова Л. А.*
Верстка *Луценко С. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16.
Гарнитура «PT Serif». Печать цифровая.
Усл. печ. л. 8,63. Тираж 200 экз.

Веб-сайт издательства: *www.dmkpress.com*

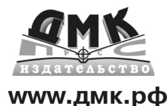
C++ – язык программирования с долгой историей и на первый взгляд непредсказуемым поведением. 25 задач помогут вам исследовать наиболее интересные его «причуды». В процессе решения за кажущейся сложностью языка вы разглядите и сможете понять фундаментальные основы его работы.

- Как на самом деле работает инициализация?
- Существуют ли временные объекты?
- Почему +!!"" — допустимое выражение в C++?

Ответы на эти и многие другие вопросы вы найдете по мере решения задач. Каждая из них представляет собой полную и с виду простую программу на C++, но сможете ли вы догадаться, что она выводит, или вам понадобится помощь? Результат с подробным объяснением приводится после каждой задачи, но не торопитесь переворачивать страницу!

Прочитав книгу, вы получите в свое распоряжение инструменты и методы, которые позволят писать более качественный и безопасный код, а также научитесь более уверенно распознавать ошибки в собственных программах.

Для изучения материала необходимо базовое знание C++. Если вы хотите запускать программы самостоятельно, то можете воспользоваться локальным компилятором C++ либо онлайн-компиляторами на сайте https://godbolt.org.



www.дмк.рф

ISBN 978-5-93700-351-5