PRO ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

bhv®

Экскурс в неопределенное поведение C++

Десятки историй о неоднозначных и сомнительных решениях в дизайне языка C++, приводящих к многим часам отладки в реальных проектах

Самые неожиданные ошибки и как они проявляются

Советы как избежать встречи с одним из самых загадочных «зверей» в разработке на C++ и других языках для системного и низкоуровневого программирования — неопределенным поведением

Дмитрий Свиридкин, Андрей Карпов

Дмитрий Свиридкин, Андрей Карпов

Экскурс в неопределенное поведение C++

Санкт-Петербург
«БХВ-Петербург»
2025

УДК 004.416
ББК 32.973.26-018.2
С24

Свиридкин, Д. О.

С24 Экскурс в неопределенное поведение C++ / Д. О. Свиридкин, А. Н. Карпов. — СПб.: БХВ-Петербург, 2025. — 384 с. — (Профессиональное программирование)

ISBN 978-5-9775-2073-7

Книга представляет собой обширный справочник типичных, а также очень редко встречающихся ошибок, характерных для программ на C++, Rust и других языках для низкоуровневого и системного программирования, в частности на ассемблере. Все рассмотренные проблемы так или иначе связаны с неопределенным, неуточненным и определяемым реализацией поведением языковых конструкций. Наибольшее внимание уделено неопределенному поведению, возможным признакам его присутствия в программах и методам поиска, диагностики и устранения такого поведения.

Для специалистов по C++ и другим языкам
для системного и низкоуровневого программирования

УДК 004.416
ББК 32.973.26-018.2

Группа подготовки издания:

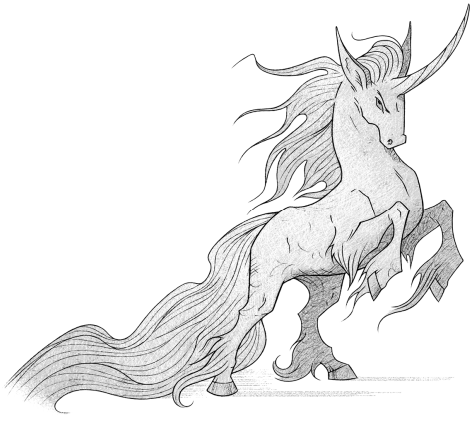
Руководитель проекта	*Олег Сивченко*
Зав. редакцией	*Людмила Гауль*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Светлана Крутоярова*
Оформление обложки	*Зои Канторович*

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

ISBN 978-5-9775-2073-7

Содержание

От автора, или Коротко о том, зачем и почему



Паникуй!

Всё начинается просто и незатейливо: обычный десятиклассник увлекается программированием, знакомится с алгоритмическими задачками, решения которых должны быть быстрыми. Узнает о языке C++, учит минимальный синтаксис, основные конструкции, контейнеры, решает задачи с предопределенным и всегда корректным форматом ввода и вывода и го́ря не знает...

В это же время где-то в большом мире матерые разработчики каждый день ругают то одни языки программирования, то другие. По самым разным причинам: неудобно, нет какой-то возможности, много лишних букв писать, ошибки в стандартной библиотеке... Но есть язык, который ругают за всё, и особенно за такую непонятную и таинственную вещь, как неопределенное поведение (undefined behavior, UB).

Спустя лет пять или шесть наш рядовой десятиклассник, блаженствующий в идеальном мире оторванных от реальности программ, внезапно узнает, что тем самым горячо нелюбимым языком всегда был, остается и будет его C++. А в дальнейшем еще в течение нескольких лет он испытает самые кошмарные и невероятные ужасы, поджидающие программистов на C++ почти на каждом шагу. Так и появится эта серия заметок, собирающая наиболее отвратительные примеры, на которые очень легко наткнуться при решении повседневных задач.

"Преждевременная оптимизация — корень всех зол" (Д. Кнут или Э. Хоар — в зависимости от того, какой источник смотрите). Язык C++ — пожалуй, наиболее яркая тому демонстрация: огромное количество ошибок в программах на C++ связано с неопределенным поведением, заложенным в фундаменте языка просто для того, чтобы дать простор оптимизациям на этапе компиляции.

Если вы собираетесь писать на C++ код, в работоспособности которого хотите быть хоть немного уверенными, сто́ит знать о существовании различных подводных камней и ловко расставленных мин в стандарте языка, его библиотеке, и всячески их избегать. Иначе ваши программы будут работать правильно только на конкретной машине и только по воле случая.

В этой книге я собрал множество самых разных примеров о том, как в коде на C и C++ можно наткнуться на неопределенное, неожиданное и совершенно ошибочное поведение. Многие из них я непосредственно встречал в проектах, над которыми мне довелось работать за последние 8 лет. И хотя основной фокус книги наведен всё же на неопределенное поведение, в некоторых разделах описываются аспекты вполне специфицированные, но довольно неочевидные.

Важно!

Эта книга **не является учебным пособием** по языку и рассчитана на тех, кто уже знаком с программированием вообще и с C++ в частности и понимает основные конструкции языка.

От научного редактора

Для того чтобы был понятен мой интерес к теме неопределенного поведения (undefined behavior, UB), начнём с того, чем я занимаюсь.

Меня зовут Андрей Карпов, и я... C++-программист. По крайней мере, точно им был. Сейчас моя деятельность сместилась в сторону DevRel-активностей и обучения сотрудников. Однако я успел вдоволь попрограммировать системы обработки больших массивов данных, поразрабатывать специализированные CAD-системы для медицины. И, самое главное, я стал одним из основателей проекта PVS-Studio — статического анализатора для поиска ошибок в коде. На момент написания книги инструмент поддерживает анализ программ на C, C++, C# и Java, а начиналось всё именно с C и C++.

Если на предыдущих местах работы я успел не раз наступить на различные грабли написания C++-кода, то, начав работать над PVS-Studio, я познал всю красоту кошмаров, которые можно создать на этом языке. Можно сказать, я узрел бездну, пучину... воплощение вселенского ужаса! И думаю, что не открою ничего неожиданного, если скажу: существенная часть этого ужаса связана как раз с неопределенным поведением.

Большинство неалгоритмических ошибок в коде на C++ связано с неопределенным поведением. Выход за границу массива/переполнение буфера — UB. Неправильное использование `printf` — UB. Деление на ноль — UB. Разыменование нулевого указателя — UB. Использование уничтоженного объекта/освобожденной памяти — UB. Переполнение переменной типа `int` — UB. Применили оператор `delete` к указателю типа `void *`? UB. Добавили отсебятину в пространства имен `std`? UB. Продолжать можно долго, доказательством чему является эта книга.

Выполняя статический анализ, нет смысла заваливать пользователя предупреждениями с упоминанием неопределенного поведения. Собственно, тогда половину предупреждений можно сформулировать как "у вас тут UB". Толку от такой формальности нет. Поэтому анализаторы сообщают о разыменовании нулевых указателей: о том, что индекс вышел за границу массива, что хорошего не будет от вот этой штуки, которую вы засунули в пространство имен `std`. Однако за всеми этими проблемами просматривается многоликое UB. Так что разработчики статических анализаторов обязаны регулярно медитировать над теорией неопределенного поведения.

Конечно, у анализатора есть и другие задачи, например, поиск опечаток или недостижимого кода. Однако UB — самый большой и неиссякаемый источник проблем в C++-программах, а соответственно, и поводов создавать новые диагностические продукты для их выявления. Поэтому я перманентно посматриваю просторы Интернета в поиске интересных материалов по этой теме, чтобы узнать, как еще можно улучшить анализатор и помочь людям избежать коварных ошибок в их коде.

Так я набрел на электронную подборку Дмитрия Свиридкина на GitHub, посвященную неопределенному поведению (ubbook). С большим любопытством с ней познакомился, выписал для себя ряд интересных мыслей, которые со временем станут основой новых диагностических правил. В общем, я получил от чтения и удовольствие, и пользу.

Затем я задумался. Во-первых, у меня тоже есть что сказать на тему неопределенного поведения. Во-вторых, таким ценным и интересным материалом хочется поделиться как можно с бóльшим количеством программистов. Это просто обидно, что такой полезный материал неизвестен широкому кругу читателей.

Я связался с Дмитрием и предложил сотрудничество по редактированию, дополнению и оформлению его публикации в книжное издание. Собственно, у него самого уже были мысли оформить всё это в виде печатной книги, так что долго мне его убеждать не пришлось. Я, скорее, выступил в качестве катализатора процесса.

Мы проработали, расширили материал, параллельно публикуя его в блоге PVS-Studio. А теперь предлагаем вашему вниманию печатный вариант книги. Думаю, получилось хорошо, и вы не раз восхититесь различными нюансами языка C++ и ловушками, которые он расставил. Дмитрий обладает большим теоретическим и практическим опытом и проделал поистине грандиозную работу, оформив свои знания в виде книги. Обещаю, будет интересно. Запасайтесь печеньками и вниманием для приятного и вдумчивого чтения.

С уважением,
Андрей Карпов

Введение

Что такое неопределенное поведение и как оно проявляется

Неопределенное поведение (undefined behavior, UB) — это удивительная особенность некоторых языков программирования, позволяющая написать синтаксически корректную программу, работающую совершенно непредсказуемо при переносе ее с одной платформы на другую, изменении опций компиляции/интерпретации и замене одного компилятора/интерпретатора другим. И главное — помимо синтаксической корректности, программа **выглядит** корректной семантически.

Состоит эта особенность в том, что в спецификации языка программирования сознательно не определяют поведение программы в каких-то особых условиях. Делается это из соображений производительности: не надо генерировать дополнительные инструкции с проверками, — или из соображений обеспечения гибкости при реализации каких-то фич. В спецификации пишут просто: "Если код делает что-то нехорошее, то поведение не определено". Например:

- если обратиться по нулевому указателю, поведение не определено;
- если дважды захватить блокировку в одном и том же потоке, поведение не определено;
- если поделить на ноль, поведение не определено;
- если прочитать неинициализированную память, поведение не определено,

  и так далее и тому подобное.

Важно, что это "поведение не определено" означает, что произойти может что угодно: форматирование диска, ошибка компиляции, исключение, а может, и все будет хорошо. Нет никаких гарантий. Отсюда и происходят неожиданные и порой очень неприятные последствия в production-коде.

И, конечно же, именно C и C++ наиболее печально известны своим неопределенным поведением. Однако надо понимать, что эта особенность присуща и другим языкам. Во многих языках можно найти какой-нибудь редкий особенный пример с неопределенным поведением. Но именно в C и C++ оно встречается при написании почти любой программы. Слишком много фич языка содержат пункты с неопределенным поведением.

Итак, по каким же признакам можно заподозрить UB в программе и насколько неопределенное поведение действительно является неопределенным?

Когда-то давно UB в коде могло повлечь действительно что угодно. Например, *gcc 1.17* начинал запускать игрушечки[1].

Сегодня, если вы поделите что-то на ноль, подобного почти наверняка не произойдет. Однако неприятности все же бывают разные:

1. Для данной конкретной платформы и компилятора в документации сказано, что именно произойдет, несмотря на страшные слова *"undefined behavior"* в стандарте. И все будет хорошо. Вы знаете что делаете. Никакой неопределенности. Все классно.
2. UB при работе с памятью чаще всего заканчиваются ошибкой сегментации и получением прекрасного сигнала `SIGSEGV` от операционной системы. Программа падает.
3. Программа работает и штатно завершается, но дает разные или неадекватные результаты от запуска к запуску. Также результаты меняются от сборки к сборке при изменении опций компилятора или самого компилятора. Никаких генераторов случайных чисел вы не использовали.
4. Программа ведет себя неправильно несмотря на то, что в коде размещено огромное множество проверок, `assert`'ов, `try-catch`-блоков, каждый из которых "подтверждает", что все корректно. В отладчике видно, что вычисления идут корректно, но совершенно внезапно все ломается.
5. Программа выполняет код, который в ней есть, но не вызывался. Отрабатывают ни разу не вызываемые функции.
6. Компилятор "без причины" и без падения отказывается собирать код. Линковщик выдает "невозможные и бессмысленные" ошибки.
7. Проверки в коде перестают исполняться. При поиске ошибок с помощью отладчика видно, что исполнение не заходит внутрь веток `if` или `catch`, хотя по значениям переменных заход должен быть выполнен.
8. Внезапный необоснованный вызов `std::terminate`.
9. Бесконечные циклы становятся конечными, и наоборот.

С неопределенным поведением часто путают другие понятия:

1. Еще одна страшная аббревиатура UB — неуточненное (*unspecified*) поведение. Стандарт не уточняет, что именно может произойти, но описывает варианты. Так, например, порядок вычисления аргументов функции — поведение неуточненное.
2. Поведение, определяемое реализацией (*implementation-defined*) — надо смотреть документацию для вашей платформы и вашего компилятора.
3. Ошибочное поведение (*erroneous*) — новинка C++26. Часть неопределенного поведения будет, возможно, переквалифицирована в эту категорию. Например,

[1] Например, ханойские башни. Подробнее об этой пасхалке см. на **https://feross.org/gcc-ownage**.

так поступили с чтением неинициализированных переменных. Разница с неопределенным — компилятору очень рекомендуется выдавать диагностики и запрещается выполнять умные оптимизации с неожиданными побочными эффектами.

Эта тройка намного лучше неопределенного поведения, хотя и имеет с ним одну общую черту: программа, полагающаяся на любое из них, вообще говоря, непереносима.

Также выделяют два класса неопределенного поведения.

- Неопределенное поведение на уровне библиотеки (*library undefined behavior*): вы сделали что-то, что не предусматривается конкретной библиотекой (в том числе и стандартной, но не всегда). Например, библиотека GMock[2] под страхом неопределенного поведения не допускает донастраивать mock-объект после начала его использования.
- Неопределенное поведение на уровне языка (*language undefined behavior*): вы сделали что-то, что фундаментально не определено спецификацией языка программирования, например, разыменовали нулевой указатель.

Если вы столкнулись с первым — у вас проблемы; но если всё работает, то с очень большим шансом и продолжит работать, пока вы не обновите библиотеку или не смените платформу. А побочные эффекты часто могут быть лишь локальными. Очень похоже на поведение implementation defined.

Если вы столкнулись со вторым — у вас большие проблемы. Код может перестать работать корректно совершенно внезапно даже при малейших изменениях. А также могут возникнуть серьезные угрозы безопасности для пользователей вашего приложения.

Как искать неопределенное поведение?

Очень частный вопрос, который задавали мне; задавал его я и сам себе, и другим. Да и каждый C++-разработчик, к сожалению, должен его задавать.

Ответ на него в общем случае — "Никак". Это алгоритмически неразрешимая задача, практически ничем не отличающаяся от задачи останова. Но программистов, как палками ни гоняй, они все равно будут решать неразрешимые задачи, так что для конкретного кода и для конкретных входных данных иногда есть способы дать ответ.

Можно проверить код до компиляции различными статическими анализаторами:

- Cppcheck;
- Clang Static Analyzer;
- PVS-Studio

и другими.

[2] GMock — популярная библиотека для тестирования. С ней мы еще встретимся!

Достаточно умный анализатор, работающий с графом потока выполнения программы, знающий сотни ловушек стандарта, способен найти многие проблемы и привлечь внимание к сомнительному коду. Но не все и не всегда.

Компиляторы *Clang* и *GCC* с включенными флагами `-Wall -Wpedantic` способны находить некоторые ошибки.

Например, GCC пожалуется на этот код:

```
int arr[5] = {1,2,3,4,5};

int main() {
    int i = 5;
    return arr[i];
}
```

выдав сообщение:

```
array subscript 5 is above array bounds of 'int [5]' [-Warray-bounds]
    6 |     return arr[i];
      |            ~~~~~^
note: while referencing 'arr'
    2 | int arr[5] = {1,2,3,4,5};
```

Мы можем сами проверять часть кода в compile-time на различных наборах входных данных, используя `constexpr`. В контексте, вычисляемом на этапе компиляции, UB запрещено[3]:

```
constexpr int my_div(int a, int b) {
    return a / b;
}

namespace test {
template <unsigned int N>
constexpr int div_test(const int (&A)[N], const int (&B)[N]) {
    int x = 0;
    for (auto i = 0u; i < N; ++i) {
        x = ::my_div(A[i], B[i]);
    }
    return x;
}
```

[3] Yaghmour Shafik "Exploring Undefined Behavior Using Constexpr" (**https://clck.ru/3JRDyo**).

```
constexpr int A[] = {1,2,3,4,5};
constexpr int B[] = {1,2,3,4,0};
static_assert((div_test(A, A), true)); // OK.
static_assert((div_test(A, B), true)); // Ошибка компиляции, деление на ноль.
}
```

Но `constexpr` не везде применим, в зависимости от версии стандарта он налагает ограничения на тело функции, а также неявно применяет `inline`-спецификатор, "запрещая" отрывать определение функции в отдельную единицу трансляции (или, попростому, определение придется разместить в заголовочном файле).

Наконец, если мы не смогли найти ошибки статическим анализом (внешними утилитами или компилятором), можно прибегнуть к динамическому анализу.

При сборке компиляторами *Clang* или *GCC* можно включить санитайзеры `-fsanitize=undefined`, `-fsanitize=address`, `-fsanitize=thread`, позволяющие отлавливать ошибки во время работы программы (run-time), но ценой значительных накладных расходов, так что пользоваться этими средствами следует лишь на этапе тестирования и разработки.

Также для отладочных сборок код стандартных библиотек иногда инструментирован `assert`'ами. Так, например, сделано для различных итераторов стандартной библиотеки в поставке MSVC[4].

Поскольку неопределенное поведение проявляется в возможностях оптимизации тем или иным компилятором, нужно собирать свой код под разные платформы, с разными уровнями оптимизаций и сравнивать его поведение. Код без ошибок должен быть переносимым и вести себя одинаково (если, конечно, его задача не генерировать совершенно случайные значения).

Тесты, различные сборки, статический и динамический анализ — способы вселить в вас уверенность в том, что в вашем коде нет UB. Дать же абсолютную гарантию может только коллегия экспертов, которые будут сверять каждую строчку кода с буквой стандарта и трижды друг друга перепроверять. И даже этого может быть недостаточно.

Еще есть путь отключения каких-либо оптимизаций флагами компилятора, включающими различные нарушения стандарта (знаменитый `-fpermissive`), превращающими язык C++ во что-то совершенно иное. Но призываю вас никогда не идти этим путем. Ваш код станет непереносимым. Он перестанет быть кодом на C++. Лучше сразу перейдите на другой язык программирования.

[4] Microsoft Visual Studio.

ГЛАВА 1
Целые и вещественные числа

Переполнение целых знаковых чисел

Бóльшая часть написанного и еще не написанного кода любой программы так или иначе работает с числами. Вычисление по каким-либо формулам, увеличение или уменьшение счетчиков итераций циклов, рекурсивных вызовов, элементов контейнеров — работа с числами везде.

Компьютер не может напрямую работать с бесконечно "длинными" числами — хранить все их цифры. Как бы много оперативной памяти у нас не было, все же она конечна. Да и хранить, и обрабатывать величины, сопоставимые с числом атомов в видимой части Вселенной, — безнадежное занятие.

Тем не менее при выполнении операций над целыми числами мы все же имеем шанс выпасть за пределы допустимого диапазона (например, $[-2^{31}; 2^{31} - 1]$ для `int32`). И тут в игру вступают особенности поддержки целых чисел для того или иного языка программирования, а также, быть может, особенности реализации конкретной платформы.

При выполнении инструкции `add` (`iadd`) платформы *x*86 переполнение целого числа сопровождается выставлением специального флага переполнения, а результирующее значение просто получается отбрасыванием старшего бита результата. И следует ожидать, что по окончании работы условной программы

```
x = 2^31 - 1
iadd x 5
```

произойдет перенос разряда в знаковый бит, и переменная `x` примет отрицательное значение.

В реализации конкретного языка программирования возможны проверка флага переполнения и сообщение об ошибке. А может и не быть. Может быть гарантия "цикличности" значений (после $2^{31} - 1$ следует -2^{31}), а может и не быть.

Проверки и гарантии — это дополнительные инструкции, которые нужно генерировать компилятору, а процессору потом исполнять.

В языке C++ решили не жертвовать производительностью и заставлять компиляторы генерировать код проверки, а объявили переполнение целых знаковых (`signed`) чисел неопределенным, открывая простор для оптимизаций. Компилятор может

генерировать любой код, какой ему вздумается, ориентируясь лишь на одно правило: *переполнения не бывает*.

Многие программисты свято верят, что переполнение чисел работает, как ожидается, "циклично", и пишут проверки вида:

```
if (x > 0 && a > 0 && x + a <= 0) {
  // обработай переполнение
}
```

Но, увы, это неопределенное поведение. И компилятор имеет полное право выкинуть такую проверку!

Пример генерации кода компилятором GCC 10.2 для *x*86–*x*64 (`-std=c++14 -O3`).

```
int main() {
  int x = 2'000'000'000;
  int y = 0;
  std::cin >> y;

  if (x > 0 && y > 0 && x + y <=0) {
    return 5;
  }
  return 0;
}
```

Обратите внимание, что в ассемблерном коде после вызова функции чтения из потока (`call`) сразу следует обнуление регистра `eax` (`xor eax, eax`) и возвращение его как результата функции.

```
main:
    sub     rsp, 24
    mov     edi, OFFSET FLAT:std::cin
    lea     rsi, [rsp+12]
    mov     DWORD PTR [rsp+12], 0
    call    std::basic_istream<char, std::char_traits<char> >::operator>>(int&)
    xor     eax, eax
    add     rsp, 24
    ret
```

Искусственный пример может быть недостаточно убедительным, так что обратим внимание на следующую — вполне серьезную — функцию вычисления полиномиального хеша строки:

```
int hash_code(std::string s) {
  int h = 13;
  for (char c : s) {
```

```
    h += h * 27752 + c;
  }
  if (h < 0) h += std::numeric_limits<int>::max();
  return h;
}
```

Функция, которая по задумке никогда не должна возвращать отрицательные числа, все-таки выдает отрицательное число! Из-за неопределенного поведения и бессмысленной с точки зрения компилятора проверки.

Компилятор может руководствоваться следующей логикой:

- если значение `h` положительно, то независимо от символа `c` величина `h*27752 + c` будет положительной: величина `c` мала, а переполнения не бывает;
- на первой итерации `h` положительно, мы суммируем положительные числа, переполнений в корректной программе не бывает, значит, на каждой итерации значение будет оставаться положительным;
- конечная сумма в итоге должна получиться положительной, и проверка не нужна.

Так и происходит при компиляции с `` `gcc-10 -O3 -std=c++17` ``:

```
std::cout << hash_code("bye");
```

Результат: `-348700627`.

Другой замечательный, но искусственный пример, для большего устрашения: конечный цикл может стать бесконечным! Пример взят из публикации "Shocking Examples of Undefined Behaviour"[1]:

```
int main() {
  char buf[50] = "y";
  for (int j = 0; j < 9; ++j) {
    std::cout << (j * 0x20000001) << std::endl;
    if (buf[0] == 'x') break;
  }
}
```

Компилятор выполняет удивительную оптимизацию умножения константы на последовательные числа, полностью изменяя заголовок цикла и условие остановки:

```
for(int j = 0; j < 9*0x20000001; j += 0x20000001) {
  ...
}
```

Условие `j < 9*0x20000001` всегда истинно, т. к. правая часть больше, чем `std::numeric_limits<int>::max()`.

С современными версиями компиляторов этот пример особенно занятен. GCC в подобных циклах иногда способен заметить переполнение и выдать предупрежде-

[1] См. блог Мохита Сани: **https://mohitmv.github.io/blog/Shocking-Undefined-Behaviour-In-Action/**.

ние. Но этого не произошло... Однако если мы закомментируем недостижимый break и buf, то получим предупреждение:

```
<source>:6:37: warning:
iteration 4 invokes undefined behavior [-Waggressive-loop-optimizations]
    6 |         std::cout << (j * 0x20000001) << std::endl;
      |                                       ^
<source>:5:23: note: within this loop
    5 |     for (int j = 0; j < 9; ++j) {
```

Если раскомментировать объявление buf, то предупреждение пропадет (GCC 13.2).

Бывает и наоборот. Ждешь последствия от переполнения, а его нет, и код магическим образом работает. Пример из статьи "Undefined behavior ближе, чем вы думаете"[2]:

```
size_t Count = size_t(5) * 1024 * 1024 * 1024; // 5 Гб
char *array = (char *)malloc(Count);
memset(array, 0, Count);

int index = 0;
for (size_t i = 0; i != Count; i++)
  array[index++] = char(i) | 1;
```

Инкрементируясь, 32-битная знаковая переменная index в какой-то момент переполнится и, кажется, должна стать отрицательной. После чего произойдёт Access Violation при выходе за границу массива. Но в случае UB никто никому ничего не должен.

Компилятор решает в целях оптимизации использовать для переменной index 64-битный регистр, который отлично увеличивается, и все элементы массива успешно заполняются. И он в своём праве: если переполнение не должно возникать, то и использовать 32-битный регистр для индекса он не обязан.

Другой, возможно, более известный и иногда полезный пример оптимизации, которую такое неопределённое поведение упрощает для компилятора — сворачивать известные суммы.

Например, при суммировании арифметических прогрессий и некоторых других известных рядов Clang 12 генерирует совершенно разный код для знаковых и беззнаковых чисел.

Вариант со знаковыми типами:

```
// суммируем квадраты от 1 до N
int64_t summate_squares(int64_t n) {
  int64_t sum = 0;
  for (int64_t i = 1; i <= n; ++i) {
    sum += i * i;
```

[2] См. **https://habr.com/ru/companies/pvs-studio/articles/276657/**.

```
  };
  return sum;
}
```

А вот ассемблерный листинг (*x*86–64 Clang 12.0.1, `-std=c++20 -O3`). Обратите внимание, что здесь нет цикла. Используется известная формула

$$(N \times (N + 1)) \times (2N + 1) / 6,$$

но довольно сложным способом:

```
summate_squares(long):                  # @summate_squares(long)
        test    rdi, rdi
        jle     .LBB2_1
        lea     rax, [rdi - 1]
        lea     rcx, [rdi - 2]
        mul     rcx
        mov     r8, rax
        mov     rsi, rdx
        lea     rcx, [rdi - 3]
        mul     rcx
        imul    ecx, esi
        add     edx, ecx
        shld    rdx, rax, 63
        movabs  rax, 6148914691236517206
        shld    rsi, r8, 63
        imul    rax, rdx
        lea     rcx, [rsi + 4*rsi]
        add     rcx, rax
        lea     rax, [rcx + 4*rdi]
        add     rax, -3
        ret
.LBB2_1:
        xor     eax, eax
        ret
*/
```

Вариант с беззнаковыми типами:

```
uint64_t usummate_squares(uint64_t n) {
  uint64_t sum = 0;
  for (uint64_t i = 1; i <= n; ++i) {
    sum += i * i;
  };
  return sum;
}
```

Здесь цикл есть. Переполнение беззнаковых типов определено и требует обработки:

```
usummate_squares(unsigned long):       # @usummate_squares(unsigned long)
        test    rdi, rdi
        je      .LBB3_1
        mov     ecx, 1
        xor     eax, eax
.LBB3_4:                               # =>This Inner Loop Header: Depth=1
        mov     rdx, rcx
        imul    rdx, rcx
        add     rax, rdx
        add     rcx, 1
        cmp     rcx, rdi
        jbe     .LBB3_4
        ret
.LBB3_1:
        xor     eax, eax
        ret
```

GCC 13 на момент написания текста (2025 год) в принципе не делает таких оптимизаций по умолчанию. При этом последние версии Clang 18 уже способны свернуть цикл суммирования квадратов и для беззнаковых[3]:

```
usummate_squares(unsigned long):       # @usummate_squares(unsigned long)
        test    rdi, rdi
        je      .LBB3_1
        inc     rdi
        cmp     rdi, 3
        mov     r8d, 2
        cmovae  r8, rdi
        lea     rax, [r8 - 2]
        lea     rcx, [r8 - 3]
        mul     rcx
        mov     rsi, rax
        mov     rcx, rdx
        lea     rdi, [r8 - 4]
        mul     rdi
        imul    edi, ecx
        add     edx, edi
        shld    rdx, rax, 63
        movabs  rax, 6148914691236517206
        shld    rcx, rsi, 63
```

[3] Читатели, искушенные в теории колец вычетов, могут для беззнаковой версии написать более простой и короткий ассемблерный код в качестве упражнения. Нужно лишь правильно поделить на 6.

```
        imul    rax, rdx
        lea     rcx, [rcx + 4*rcx]
        add     rcx, rax
        lea     rax, [rcx + 4*r8]
        add     rax, -7
        ret
.LBB3_1:
        xor     eax, eax
        ret
```

Корректные проверки переполнения в арифметических операциях намного сложнее, чем просто отслеживание смены знака.

Так, для C++20 безопасный обобщенный код арифметических операций над целыми знаковыми числами мог бы выглядеть следующим образом:

```
#include <concepts>
#include <type_traits>
#include <variant>
#include <limits>

namespace safe {

// Все эти проверки справедливы только для целых знаковых чисел.
template <class T>
concept SignedInteger = std::is_signed_v<T>
                     && std::is_integral_v<T>;

enum class ArithmeticError {
    Overflow,
    ZeroDivision
};

template <SignedInteger I>
using ErrorOrInteger = std::variant<I, ArithmeticError>;

template <SignedInteger I>
ErrorOrInteger<I> add(I a,    // выключаем вывод параметра шаблона по
                      std::type_identity_t<I> b) // второму аргументу
{
  if (b > 0 && a > std::numeric_limits<I>::max() - b) {
    // положительное переполнение
    return ArithmeticError::Overflow;
  }
```

```
  if (b < 0 && a < std::numeric_limits<I>::min() - b) {
    // отрицательное переполнение
    return ArithmeticError::Overflow;
  }
  return a + b;
}

template <SignedInteger I>
ErrorOrInteger<I> sub(I a, std::type_identity_t<I> b) {
  if (b < 0 && a > std::numeric_limits<I>::max() + b) {
    // положительное переполнение
    return ArithmeticError::Overflow;
  }
  if (b > 0 && a < std::numeric_limits<I>::min() + b) {
    // отрицательное переполнение
    return ArithmeticError::Overflow;
  }
  return a - b;
}

template <SignedInteger I>
ErrorOrInteger<I> mul(I a, std::type_identity_t<I> b) {
  if (a == 0 || b == 0) {
    return 0;
  }

  if (a > 0) {
    if (b > 0) {
      if (a > std::numeric_limits<I>::max() / b) {
        return ArithmeticError::Overflow;
      }
    } else {
        if (b < std::numeric_limits<I>::min() / a) {
          return ArithmeticError::Overflow;
        }
    }
  } else {
      if (b > 0) {
        if (a < std::numeric_limits<I>::min() / b) {
          return ArithmeticError::Overflow;
      }
```

```
      } else {
          if (b < std::numeric_limits<I>::max() / a) {
            return ArithmeticError::Overflow;
      }
    }
  }
  return a * b;
}

template <SignedInteger I>
ErrorOrInteger<I> div(I a, std::type_identity_t<I> b) {
  if (b == 0) {
      return ArithmeticError::ZeroDivision;
  }

  if (a == std::numeric_limits<I>::min() && b == -1) {
      // Диапазон [min, max] несимметричный относительно 0.
      // abs(min) > max - будет переполнение.
      return ArithmeticError::Overflow;
  }
  return a / b;
}


template <SignedInteger I>
ErrorOrInteger<I> mod(I a, std::type_identity_t<I> b) {
  if (b == 0) {
      return ArithmeticError::ZeroDivision;
  }

  if (b == -1) {
      // По стандарту в этом случае также неопределенное поведение при
      // a == std::numeric_limits<I>::min()
      // поскольку остаток и неполное частное от деления,
      // например, на платформе x86
      // получаются одной и той же инструкцией div (idiv),
      // что потребует дополнительной обработки.
      //
      // Но совершенно ясно, что остаток от деления чего угодно на -1 равен 0.
```

```
      return 0;
  }
  return a % b;
}

}
```

Если вам не нравится возвращать ошибку или результат, можете использовать *исключения*.

Видно, что безопасные версии арифметических операций должны быть как минимум в два раза медленнее своих исходно небезопасных версий. Такая экономия тактов может быть оправдана, если вы разрабатываете, например, математическую библиотеку, и вся ваша производительность упирается в CPU и перемалывание чисел.

Однако если ваша программа только и делает, что ожидает и выполняет IO операции, то траты в два раза большего числа тактов на сложение или умножение никто и не заметит. Да и язык C++ для таких программ чаще всего не лучший выбор.

Итак, если вы работаете только лишь с беззнаковыми числами (`unsigned`), то с неопределенным поведением при переполнении никаких проблем нет: всё определено как вычисления по модулю[4] 2^N.

Если же вы работаете со знаковыми числами, то либо используйте безопасные обертки, сообщающие каким-либо образом об ошибках, либо выводите ограничения на входные данные программы целиком таким образом, чтобы переполнения не возникало, и не забывайте эти ограничения проверять. Либо компилируйте с флагом `-fwrapv`[5]. Всё просто, да[6]?

Для выведения ограничений вам помогут отладочные `assert` с правильными проверками переполнения, которые нужно написать. Или включение `ubsan` (*undefined behavior sanitizer*) при сборке компиляторами Clang или GCC. А также тестовые `constexpr`-вычисления.

Также проблемы неопределенного поведения при переполнении касаются битовых сдвигов влево для отрицательных чисел (или при сдвиге положительного числа с залезанием в знаковый бит). Начиная с C++20, стандарт требует фиксированной единой реализации отрицательных чисел — через дополнительный код (two's complement), и многие проблемы сдвигов сняты. Тем не менее все равно стоит следовать общей рекомендации: любые битовые операции выполнять только в `unsigned`-типах.

---

[4] N — количество битов для выбранного типа чисел. Например, 64 для `uint64_t`.

[5] Этот флаг для GCC и Clang форсирует циклическое поведение (wrapping) при переполнении.

[6] В C++26 у вас еще будут на выбор операции для арифметики с насыщением: `std::add_sat`, `std::sub_sat`, `std::mul_sat` и `std::div_sat`.

Дополнительный код (two's complement)

Дополнительный код — наиболее распространенный способ представления отрицательных целых чисел в компьютерах. Он позволяет заменить операцию вычитания операцией сложения и сделать операции сложения и вычитания одинаковыми для знаковых и беззнаковых чисел.

Десятичное представление	Двоичное представление (8 бит)
127	0111 1111
1	0000 0001
0	0000 0000
-0	нет
-1	1111 1111
-2	1111 1110
-3	1111 1101
-8	1111 1000
-10	1111 0110
-127	1000 0001
-128	1000 0000

Дополнительный код для отрицательного числа можно получить инвертированием его двоичного модуля (получается "первое дополнение") и прибавлением к инверсии единицы (получается "второе дополнение").

Дополнительный код двоичного числа определяется как величина, полученная вычитанием числа из наибольшей степени двух.

Стоит заметить, что сужающее преобразование из целочисленного типа в другой целочисленный тип к неопределенному поведению не приводит, и выполнять побитовое И с маской перед присваиванием переменной меньшего типа необязательно. Но желательно, чтобы избежать предупреждений компилятора:

```
constexpr int x = 12345678;
constexpr uint8_t first_byte = x; // предупреждение о неявном приведении
```

Очень неприятным является переполнение целочисленных переменных, возникающее из-за правил *integer promotion*:

```
constexpr std::uint16_t IntegerPromotionUB(std::uint16_t x) {
    x *= x;
    return x;
}

// 65535 * 65535 mod 1<<16 = 1

static_assert(IntegerPromotionUB(65535) == 1); // Не скомпилируется!
```

Несмотря на то что для беззнаковых типов переполнение определено как взятие остатка по модулю 2^N и мы используем только беззнаковую переменную, из-за *integer promotion* здесь неявно возникает переполнение знакового (!) числа и вытекающее из этого UB.

Справедливости ради надо заметить, что такое происходит лишь на платформах, где размер `int` больше `uint16_t`[7]:

```
x *= x; // переписывается как x = x * x;
```

Тип `uint16` меньше, чем тип `int`. Для умножения выполняется неявное приведение к `int`.

Сужающие преобразования и неявное приведение типов

Неявные преобразования типов запрещены во многих современных языках программирования, особенно новых.

В частности, в Rust, Haskell, Kotlin нельзя просто так использовать `float` и `int` в одном арифметическом выражении без явного указания преобразовать одно в другое. Python не так строг, но все же не дает смешивать строки, символы и числа.

В C++ запрета неявного преобразования нет, что порождает массу ошибочного кода. Причем в таком коде может быть как определенное, но неожиданное, так неопределенное поведение.

Пример:

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

int average(const std::vector<int>& v) {
    if (v.empty()) {
        return 0;
    }
    return std::accumulate(v.begin(), v.end(), 0) / v.size();
}

int main() {
    std::cout << average({-1,-1,-1});
}
```

Любой, кто мельком бросит взгляд на этот код, будет ожидать, что результатом работы окажется –1. Но, увы, результат будет совершенно другим. Программа, собранная GCC под платформу *x*86–*x*64, распечатает:

```
1431655764
```

[7] То есть практически везде в наши дни. 64-битная эпоха все-таки.

В этом коде нет неопределенного поведения (по крайней мере, на используемых входных данных). Но есть неявное приведение типов, делающее результат неожиданным.

1. Тип возвращаемого значения `std::accumulate` определяется третьим аргументом. В данном случае это целочисленный знаковый ноль — тип по умолчанию для всех числовых литералов.
2. Тип возвращаемого значения операции деления определяется наибольшим из участвующих типов аргументов, а также правилами *integer promotion*. В примере тип левого аргумента — `int`, а правого — `size_t`, достаточно широкое беззнаковое целое, более широкое, чем `int`. Потому, по правилам *integer promotion*, результатом будет `size_t`.
3. –3 неявно преобразуется к типу `size_t` — такое преобразование вполне определено. Результатом будет беззнаковое число $2^N - 3$.
4. Далее будет произведено деление беззнаковых чисел: $(2^N - 3) / 3$. Старший бит результата окажется нулевым.
5. Возвращаемым типом функции `average` объявлен `int`. Так что нужно выполнить еще одно неявное преобразование.
6. В общем случае преобразование `unsigned -> signed` определяется реализацией (implementation defined).
   - Если размеры типов `int` и `size_t` одинаковые, то, поскольку старший бит нулевой, положительное число укладывается в допустимый диапазон значений для типа `int` — стандарт гарантирует, что никаких проблем нет.
   - Если размеры не совпадают, то произойдет сужающее преобразование (*narrowing conversion*), которое как раз-таки отдано на откуп деталям реализации. Так, вместо ожидаемой обрезки старших непоместившихся битов на некоторых платформах может произойти замена на `std::numeric_limits<int>::max`.
   - Для примера сборки под 64-битную платформу с помощью *gcc* сужающее преобразование определено, как и ожидается, через обрезку старших битов. Поэтому итоговым результатом оказывается $((2^{64} - 3) / 3 \% 2^{32})$.

Неявные приведения типов касаются не только встроенных примитивов, но и более сложных типов. И самое неприятное — они вмешиваются в выбор подходящей перегрузки функции, приводя к различным, часто неприятным, казусам.

Пример с `abs`:

```
#include <cmath>
#include <iostream>

int main() {
    std::cout << abs(3.5) << "\n"; // функция библиотеки C,
                                   // принимает на вход тип long;
                                   // результат равен 3
```

```
    std::cout << std::abs(3.5);     // функция библиотеки C++,
                                    // перегружена для double;
                                    // результат равен 3.5
}
```

Еще более неприятный пример наблюдается со стандартным типом `std::string`:

```
#include <string>

int main() {
    std::string s;
    s += 48;    // неявное приведение к char
    s += 1000;  // а тут еще и с переполнением, очень неприятным
                // на платформе с signed char
    s += 49.5;  // опять-таки неявное приведение к char
}
```

Этот ужас компилируется!

Казалось бы, этот пример совершенно жуткого использования никогда не может встретиться в нормальном коде. Увы, но может.

Вы можете написать обобщенный код своего `std::accumulate` с различными проверками шаблонных аргументов и случайно, по ошибке, передать в него `string` в качестве аккумулятора и контейнер, например, `float`. И никакой ошибки компиляции не произойдет. Только странный баг в программе.

Эта программа компилируется, а результатом является пустая строка:

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

template <class Range, class Acc>
auto accumulate(Range&& r, Acc acc)
requires(requires(){
    {acc += *std::begin(r) };
})
{
    for (auto&& x : r){
        acc += x;
    }
    return acc;
}
```

```
int main() {
    std::vector<double> v {0.5, 0.7, 0.1};
    auto res = accumulate(v, std::string{});
    std::cout << '"' << res << '"';
}
```

Вывод программы:

```
""
```

Цепочки неявных преобразований могут быть очень неочевидными.

```
void f(float&& x) { std::cout << "float " << x << "\n";  }
void f(int&& x) { std::cout << "int " << x << "\n";  }
void g(auto&& v) { f(v); } // C++20
// template <class T> void g(T v) { f(v); }
int main() {
    g(2);
    g(1.f);
}
```

Самым удивительным образом этот пример выводит:

```
float 2
int 1
```

Хотя мы подставляли типы констант совсем наоборот и почти наверняка ожидали:

```
int 2
float 1
```

Это не баг компилятора и не неопределенное поведение! Всему виной хитрая цепочка неявных преобразований.

Рассмотрим ее на примере первого вызова `g(2)`, подставив параметр шаблона:

```
void g(int&& v) {
    // Несмотря на то что тип v - это int&&,
    // дальнейшее использование v в выражениях дает int& !
    // decltype(v)   == int&&
    // decltype((v)) == int&

    // Функции f принимают только rvalue-ссылки

    // Неявное преобразование int& к int&& запрещено
    //  int&& x = 5;
    //  int&& y = x; // не компилируется!
```

```
    // Таким образом, перегрузка f(int&&) не может быть использована

    // Остается f(float&&);
    // int умеет неявно приводиться к float;
    // int& умеет неявно выступать в роли просто int;
    // неявный static_cast<float>(v) возвращает временное значение float;
    // временные значения типа T неявно биндятся к T&&

    // Имеем цепочку преобразований:
    // int& -> int -> float -> float&&

    f(v); // будет вызван f(float&&) !

    // явно: f(static_cast<float>(v));
}
```

Конечно, никто никогда (по крайней мере явно) не принимает примитивы по *rvalue*-ссылкам, потому что это бессмысленно и неэффективно. Но даже без *rvalue*-ссылки для примитивов мы можем сотворить нечто ужасное:

```
struct MyMovableStruct {
    operator bool () {
        return !data.empty();
    }
    std::string data;
};

void consume(MyMovableStruct&& x) {
    std::cout << "MyStruct: " << x.data << "\n";
}
void consume(bool x) { std::cout << "bool " << x << "\n";  }
void g(auto&& v) { consume(v); }
int main() {
    g(MyMovableStruct{"hello"});
}
```

Той же самой цепочкой преобразований получим в выводе "`bool 1`". Разве что последний шаг не нужен.

Обязательно включайте предупреждения компилятора обо всех неявных преобразованиях. Очень желательно трактовать их как ошибки.

Не привносите неявные преобразования для своих типов — всегда помечайте однопараметрические конструкторы как `explicit`.

Если перегружаете операторы приведения (`operator T()`) для своих типов — также делайте их `explicit`.

Если ваши функции/методы рассчитаны на работу только с определенным примитивным типом, навешивайте на них ограничения с помощью шаблонов, правила SFINAE (substitution failure is not an error — сбой замены не является ошибкой), концептов, или, что очень просто, механизма явного удаления перегрузок (`= delete`):

```
int only_ints(int x) { return x;}

template <class T>
auto only_ints(T x) = delete;

int main() {
    const int& x = 2;
    only_ints(2);
    only_ints(x);
    char c = '1';
    only_ints(c);   // ошибка компиляции
    only_ints(2.5); // перегрузка явно удалена
}
```

Integer promotion

C++ от C досталось тяжелое наследство. Одна его часть была исправлена и беспощадно зарезана для большей надежности: так, например, поступили с неявными `const`-преобразованиями. Другая же часть, доставляющая не меньше проблем, перешла в первозданном виде.

В C и C++ много различных типов целых чисел разных размеров. И над ними определены операции. Правда, операции определены не для каждого типа чисел.

Например, здесь нет `+`, `-`, `*`, `/` для `uint16_t`. Но применить мы их можем, и результатом операций над беззнаковыми числами станет число со знаком.

```
uint16_t x = 1;
uint16_t y = 2;
auto a = x - y;   // a имеет тип int
auto b = x + y;   // b имеет тип int
auto c = x * y;   // c имеет тип int
auto d = x / y;   // d имеет тип int
```

Хотя это опять не вся правда. Если `int` окажется 16-битным, то `a`, `b`, `c` и `d` станут `unsigned int`. Ну и стоит тип хотя бы одного аргумента поменять на `uint32_t`, как результат сразу же теряет знак.

Что происходит?

Происходят две неявные операции.

- Типы, меньшие `int`, приводятся к `int` (integer promotion). Знаковому! Независимо от знаковости исходного типа!
- Когда в операции участвуют аргументы разных типов целых чисел, они приводятся к общему типу (usual arithmetic conversion):
  - меньший тип приводится к большему;
  - если размеры одинаковы, то знаковый приводится к беззнаковому.

Аналогичные операции проводятся и над числами с плавающей точкой. За полной таблицей и цепочкой, показывающей, что именно и во что неявно превращается, сто́ит обратиться к тексту стандарта[8].

К чему это приводит?

1. К ошибкам в логике. Неявные преобразования вовлекаются в любую операцию. Вы выполняете сравнение знакового и беззнакового чисел и забыли явно привести типы? Готовьтесь к тому, что `-1 < 1` может вернуть `false`:

```
std::vector<int> v = {1};
auto idx = -1;
if (idx < v.size()) {
    std::cout << "less!\n";
} else {
    std::cout << "oops!\n";
}
```

2. К неопределенному поведению:

```
unsigned short x=0xFFFF;
unsigned short y=0xFFFF;
auto z=x*y;
```

   Integer promotion неявно приводит `x` и `y` к `int`, в котором происходит переполнение. Переполнение `int` — неопределенное поведение.

3. К трудностям в переносе программ с одной платформы на другую. Если меняется размер `int`/`long`, то применение правил неявных конверсий к вашему коду также меняется:

```
std::cout << (-1L < 1U);
```

   Код выводит разные значения в зависимости от размера типа `long`.

Что делать?

- Не смешивать в одном выражении знаковые и беззнаковые типы.
- Уделять особое внимание коду, работающему с типами, меньшими `int`.

[8] См. § 6.8.6 "Conversion ranks" в документации (**https://eel.is/c++draft/conv.rank**).

- Включать предупреждения от компилятора (`-Wconversion`, не всегда работает).
- Посматривать на диагностические сообщения анализаторов кода.

Тип char и знаковое расширение

Возьмем следующую простенькую структуру:

```
// Пример (с изменениями) взят отсюда:
// https://twitter.com/hankadusikova/status/1626960604412928002
struct CharTable {
    static_assert(CHAR_BIT == 8);
    std::array<bool, 256> _is_whitespace {};

    CharTable() {
        _is_whitespace.fill(false);
    }

    bool is_whitespace(char c) const {
        return this->_is_whitespace[c];
    }
};
```

Всё ли в порядке с этим безобидным методом `is_whitespace`? Ну, кроме того, что `char` в C и C++ обычно восьмибитный, а в Unicode есть пробельные символы, кодируемые 16 битами[9].

Давайте потестируем:

```
int main() {
    CharTable table;
    char c = 128;
    bool is_whitespace = table.is_whitespace(c);
    std::cout << is_whitespace << "\n";
    return is_whitespace;
}
```

При сборке с `-fsanitize=undefined` получаем дивный результат:

```
/opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/array:61:36:
runtime error: index 18446744073709551488 out of bounds for type 'bool [256]'
/opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/array:61:36:
runtime error: index 18446744073709551488 out of bounds for type 'bool [256]'
/app/example.cpp:14:38:
runtime error: load of value 64, which is not a valid value for type 'bool'
```

[9] Например, такой удивительный символ, как U+FEFF — ZERO WIDTH NO-BREAK SPACE.

Конкретное значение в третьей строке совершенно случайное. Было бы очень здорово стабильно видеть 42, но увы.

Зато индекс в первых двух строках совсем не случайный.

Но погодите, `char c = 128`, а это же точно меньше 256. Откуда `18446744073709551488`?

Будем разбираться. В деле замешаны две удачно разложенные ловушки.

- Специфичная ловушка C и C++: знаковость типа `char` не специфицирована. В зависимости от платформы он может быть как знаковым, так и беззнаковым. На *x*86 чаще всего является знаковым. И из `char c = 128` получается `c = -128`.
- Ловушка, распространенная во многих языках, имеющих разные типы целых чисел разной знаковости и длины. Например, в Rust:

```
pub fn main() {
    let c : i8 = -5;
    let c_direct_cast = c as u16;
    let c_two_casts = c as u8 as u16;
    println!("{c_direct_cast} != {c_two_casts}");
}
```

  Мы увидим `65531 != 251`.

При преобразовании знакового целого меньшей длины к беззнаковому целому большей длины происходит знаковое расширение: старшие биты заполняются битом знака.

То же действует и в C и C++:

```
int main() {
    int8_t c  = -5;
    uint16_t c_direct_cast = c;
    uint16_t c_two_casts = static_cast<uint8_t>(c);
    std::cout << c_direct_cast << " != " << c_two_casts;
}
```

Напечатает `65531 != 251`.

А теперь остается только взглянуть на сигнатуру `std::array::operator[]`:

```
reference operator[]( size_type pos );
```

`size_type` — это беззнаковый `size_t`. Под *x*86 он определенно больше, чем `char`. Происходит прямое преобразование знакового `char` в `size_t`, знак расширяется, код ломается. Дело закрыто.

Что делать?

Со знаковым расширением иногда способны помочь статические анализаторы. Нужно понимать, что именно вы делаете при преобразовании чисел и что хотите получить. Часто можно встретить конструкцию вида `uint32_t extended_val = static_cast<uint32_t>(byte_val) & 0xFF`, которая гарантированно занулит верхние бай-

ты, и вы сможете избежать знакового расширения. Аналогичная конструкция может быть и при преобразовании `int32 -> uint64`, и при любых других комбинациях. Только константу правильную писать не забывайте.

Из-за своей знаковой неспецифицированности тип `char` очень опасен при работе с ним как с типом чисел. Крайне рекомендуется пользоваться соответствующими типами `uint8_t` или `int8_t`, или другими подходящими, если на вашей целевой платформе в `char` внезапно не 8 бит.

Унарный минус и беззнаковые числа

Вы разрабатываете графический интерфейс для игры. У вас уже есть кнопочки, панельки, иконки. Все отлично. И вот вы решаете, чтобы интерфейс ощущался интереснее, реализовать анимацию для элемента checkbox (флажка) — при нажатии для снятия галочки он будет красиво отъезжать в сторону где-то на 30% своей ширины.

У вас были подобные структуры и функции:

```
struct Element {
    size_t width; // немасштабированная ширина
    ...
}

using ElementID = uint64_t;
// В OpenGL/DirectX/Vulkan координаты обычно вещественные
struct Offset {
    float x;
    float y;
};

size_t get_width(ElementID);
float screen_scale();
void move_by(ElementID, Offset);
```

И вы добавили свою:

```
void on_unchecked(ElementID el) {
    auto w = get_width(el);
    move_by(el, Offset {
        -w * screen_scale() * 0.3f,
        0.0f
    });
}
```

Ваш checkbox имел ширину 50 пикселов. Вы запустили тест... И элемент улетел за пределы экрана!

Вы пошли смотреть логи и обнаружили:

```
Offset: 5.5340234e+18 0
```

Как же так?! Неопределенное поведение?

Нет. Вполне определенное.

Всему виной унарный минус, который мы случайно применили к беззнаковой переменной.

`For unsigned a, the value of -a is` 2^N `- a, where N is the number of bits after promotion.`

Это очень злобная ошибка, которую не диагностируют Clang и GCC с флагами `-Wall -Wextra -Wpedantic`; MSVC же выдает диагностику `C4146`.

Статические анализаторы, например PVS-studio (v2553), также могут найти ошибку.

В более современных языках программирования применение унарного минуса к беззнаковым значениям чаще всего не компилируется. Так, например, сделано в Rust, Zig и в Kotlin.

Числа с плавающей точкой

С `float` и `double` в принципе всегда всё сложно. Особенно в C++.

Стандарт C++ не требует следования стандарту IEEE 754, потому деление на ноль в вещественных числах также считается неопределенным поведением несмотря на то, что по IEEE 754 выражение `x/0.0` определяется как `-INF`, `NaN`, или `INF` в зависимости от знака числа `x` (`NaN` для нуля).

Сравнение вещественных чисел — излюбленная головная боль.

Выражение `x == y` фактически является кривым побитовым сравнением для чисел с плавающей точкой, по-особенному работающим со случаями `-0.0` и `+0.0`, и `NaN`. О существовании этого и `!=` операторов для вещественных чисел стоит забыть и никогда не вспоминать[10].

Для побитового сравнения нужно использовать *memcmp*. Для сравнения чисел — приближенные варианты вида `std::abs(x - y) < EPS`, где *EPS* — какое-то абсолютное или вычисляемое на основе `x` и `y` значение. А также различные манипуляции с ULP (*unit in the last place*) сравниваемых чисел[11].

Так как стандарт C++ не форсирует IEEE 754, проверки на `x == NaN` через его свойство `(x != x) == true` могут быть убраны компилятором как заведомо ложные. Проверять нужно с помощью предназначенных для этого функций `std::isnan`.

---

[10] На тот случай, если вам по наследству достался большой проект, и хочется узнать, как в нем обстоит дело со сравнением чисел с плавающей точкой, вы можете воспользоваться анализатором PVS-Studio. В нем есть диагностика `V550: Suspicious precise comparison`.

[11] См. "Unit in the last place" (**https://en.wikipedia.org/wiki/Unit_in_the_last_place**).

Поддерживается или нет IEEE 754, можно проверить с помощью предопределенной константы `std::numeric_limits<FloatType>::is_iec559`.

Сужающие преобразования из `float` в знаковые или беззнаковые целые могут повлечь неопределенное поведение, если значение непредставимо в целочисленном типе. Никаких обрезок по модулю 2^N не предполагается.

```
constexpr uint16_t x = 1234567.0; // CE, неопределенное поведение
```

Обратное преобразование (из целочисленных типов во `float`/`double`) также имеет свои подвохи, не связанные с неопределенным поведением: большие по абсолютной величине целые числа теряют точность.

```
static_assert(
  static_cast<float>(std::numeric_limits<int>::max()) ==  // OK
  static_cast<float>(static_cast<long long>(
    std::numeric_limits<int>::max()) + 1)
);

static_assert(
  static_cast<double>((1LL << 53) - 1) == static_cast<double>(1LL << 53) // Огонь!
);

static_assert(
  static_cast<double>((1LL << 54) - 1) == static_cast<double>(1LL << 54) // OK
);

static_assert(
  static_cast<double>((1LL << 55) - 1) == static_cast<double>(1LL << 55) // OK
);

static_assert(
  static_cast<double>((1LL << 56) - 1) == static_cast<double>(1LL << 56) // OK
);
```

В качестве домашнего задания попробуйте самостоятельно сформулировать, почему никогда нельзя хранить финансы в типах с плавающей запятой.

Плавающая точка и шаблоны

До C++20 вещественные числа нельзя было использовать в качестве параметров-значений в шаблонах. Теперь же можно. Правда, ожидать, что вы насчитаете в runtime и в compile-time одно и то же, не стоит.

Различие вычислений во время run-time и compile-time

Вот код, правда, на C, но суть та же. Первый вызов функции `expl`[12] разворачивается в константу, а второй по-честному вычисляется:

```
#include <stdio.h>
#include <string.h>
#include <math.h>

static void printBits(size_t const size, void const * const ptr)
{
  unsigned char *b = (unsigned char*) ptr;
  unsigned char byte;
  int i, j;

  for (i = size * 8; i > 0; i--) {
    if( i % 8 == 0)
    {
      printf("%d", i);
      if( i >= 100) i-=2;
      else if( i >= 10) i-=1;
    }
    else {printf(" ");}
  }
  printf("\n");
  for (i = size * 8; i > 0; i--) {
    if( i%8 == 0) {printf("|");} else {printf(" ");}
  }
  printf("\n");
  for (i = size-1; i >= 0; i--) {
    for (j = 7; j >= 0; j--) {
      byte = (b[i] >> j) & 1;
      printf("%u", byte);
    }
  }
  printf("\n");
}

int main()
{
  long double c, r1, r2;
```

[12] `expl(x)` — это e^x, определенный для значений типа `long double`.

```
  r1 = expl(-1);

  c = -1;
  r2 = expl(c);

  printBits(sizeof(r1), &r1);
  printBits(sizeof(r2), &r2);

  if( memcmp( &r1, &r2, sizeof(r1)) != 0 )
  {
    printf("Not equal!\n");
    return 1;
  }

  printf("Equal!\n");
  return 0;
}
```

Давайте посмотрим на результат работы кода. Различаются не только run-time- и compile-time-варианты вычислений. На результат влияют еще и ключи оптимизации!

Вывод при использовании компилятора GCC 14.1 для *x*86–*x*64:

```
128     120     112     104     96      88      80      72      64      56      48      40      32      24      16      8
|       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
00000000000000000000000000000000000000000000000011111111111101101111000101101010110001101100010110011101110011011111000110101
128     120     112     104     96      88      80      72      64      56      48      40      32      24      16      8
|       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
00000000000000000000000000000000000000000000000011111111111101101111000101101010110001101100010110011101110011011111000110110
Not equal!
```

Вывод при использовании компилятора x86–x64 GCC 14.1 с ключом `-O3`:

```
128     120     112     104     96      88      80      72      64      56      48      40      32      24      16      8
|       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
00000000000000000111011101111100110100110010100011111111111101101111000101101010110001101100010110011101110011011111000110101
128     120     112     104     96      88      80      72      64      56      48      40      32      24      16      8
|       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
00101110011110000111100001100011011000100110100100111111111111011011110001011010101100011011000101100111011100110111110001101
Not equal!
```

Для простой параметризации типов константами этот механизм вполне можно использовать без опасений. Однако строить на них паттерн-матчинг с выбором специализаций шаблонов крайне не рекомендуется:

```
template <double x>
struct X {
    static constexpr double val = x;
};
```

```
template <>
struct X<+0.> {
    static constexpr double val = 1.0;
};

template <>
struct X<-0.> {
    static constexpr double val = -1.0;
};


int main() {
    constexpr double a = -3.0;
    constexpr double b = 3.0;
    std::cout << X<a + b>::val << "\n";          // печатает +1
    std::cout << X<-1.0 * (a + b)>::val << "\n"; // печатает -1
    static_assert(a + b == -1.0 * (a + b));      // ok
}
```

По тем же причинам ни в одном языке программирования не рекомендуется использовать значения с плавающей точкой в качестве ключей ассоциативных массивов.

ГЛАВА 2

Нарушение жизненного цикла объектов

Висячие ссылки, указатели и use-after-free: общие случаи

80% случаев неопределенного поведения в C++ связаны с висячими ссылками (dangling references).

Объект жил на стеке и умер. Или объект жил в куче и умер. Разница, по сути, не очень большая: обобщенный сценарий воспроизведения ошибки один и тот же, — где-то остались указатель или ссылка на уже мертвый объект. А потом этой ссылкой (или указателем) воспользовались, чтобы обратиться к мертвому объекту. Такой спиритический сеанс заканчивается неопределенным поведением. Если повезет, будет ошибка сегментации с возможностью узнать, кто именно обратился к объекту.

Но всё же между первым объектом со стека или первым объектом из кучи есть разница в возможности обнаружения методами динамического анализа. Для инструментации стека санитайзерами, вообще говоря, нужно перекомпилировать программу. Для инструментации кучи можно подменить библиотеку с аллокатором.

Конечно, в жизни почти никто и никогда явно не пишет некорректный код вида:

```
int main() {
  int* p = nullptr;
  {
    int x = 5;
    p = &x;
  }
  return *p;
}
```

Но проблема в том, что подобный код в языке C++ может быть ловко замаскирован под слоем абстракций из классов и функций.

Простой пример:

```
int main() {
  const int x = 11;
```

```
  auto&& y = std::min(x, 10);
  std::cout << y << "\n";
}
```

В этом коде неопределенное поведение из-за висячей ссылки. Такая программа, будучи собранной, например, компилятором GCC 14.1 с ключами `-std=c++17 -O3`, не упадет, но неожиданно напечатает `0`.

Проблема в том, что `std::min` объявлен как:

```
template<class T> const T& min(const T& a, const T& b);
```

Число `10` является вре́менным объектом (`prvalue`), который умирает сразу же при выходе из функции `std::min`.

В C++ разрешено присваивать временные объекты константным ссылкам. В таком случае константная ссылка продлевает временному объекту жизнь (объект "материализуется") и живет до выхода ссылки из области видимости. Дальнейшие присваивания константным ссылкам эффекта продления времени жизни не имеют.

Любой код, возвращающий из функции или метода ссылку или сырой указатель, является потенциальным источником проблем где угодно. Код, который только принимает аргументы по ссылке и никуда эти ссылки не сохраняет, также может быть источником проблем, но в более неочевидных ситуациях.

```
template <class T>
void append_n_copies(std::vector<T>* elements,
                     const T& x, int N)
{
  for (int i = 0; i < N; ++i) {
    elements->push_back(x);
  }
}

void foo() {
  std::vector<int> v; v.push_back(10);
  ...
  append_n_copies(&v, v.front(), 5); // будет неопределенное поведение
                                     // при реаллокации вектора!
}
```

У такого кода есть все шансы появиться в реальном проекте и доставить множество неприятностей.

В некритичных к производительности участках кода лучше использовать передачу по значению и перемещение вместо передачи по ссылке. Это, увы, также не снимает всех проблем, но ошибка в программе будет явно локализована в точке вызова функции, а не размазана по ее телу.

```
template <class T>
std::vector<T> append_n_copies(std::vector<T> elements,
                               T x, int N) {
```

```
  for (int i = 0; i < N; ++i) {
    elements.push_back(x);
  }
  return elements;                // неявное перемещение
}

void foo() {
  std::vector<int> v; v.push_back(10);
  ...
  // v = append_n_copies(std::move(v), v.front(), 5);
  // UB, use-after-move, порядок вычисления аргументов не определен:
  // v.front() может быть вызван на пустом векторе

  auto el = v.front();
  v = append_n_copies(std::move(v), std::move(el), 5);
}
```

Если нужно работать со ссылками, стоит озаботиться их безопасностью.

Например, можно использовать `std::reference_wrapper`, которому нельзя присваивать временные объекты.

```
#include <utility>

template <class T>
std::reference_wrapper<const T>
  safe_min(std::reference_wrapper<const T> a,
           std::reference_wrapper<const T> b)
{
  return std::min(a, b);
}

int main() {
  const int x = 11;
  auto&& y = safe_min<int>(x, 11);   // ошибка компиляции
}
```

Или с помощью *forwarding references* проанализировать категорию (*rvalue*/*lvlaue*) переданного аргумента и решить, что с ним делать. На C++20 это выглядит так:

```
#include <type_traits>

template <class T1, class T2>
requires
  std::is_same_v<std::decay_t<T1>,
                 std::decay_t<T2>> // std::min требует одинаковых типов
```

```
decltype(auto) // выводим тип без отбрасывания ссылок
safe_min(T1&& a, T2&& b) // передаваемая ссылка на каждый аргумент
{
  if constexpr (std::is_lvalue_reference_v<decltype(a)> &&
                std::is_lvalue_reference_v<decltype(b)>) {
      // оба аргумента были lvalue, можно безопасно вернуть ссылку
      return std::min(a, b);
  } else {
    // один из аргументов - временный объект,
    // возвращаем по значению,
    // для этого делаем копию
    auto temp = std::min(a,b); // auto&& нельзя!
                               // иначе return выведет ссылку
    return temp;
  }
}
```

Конкретно для функций `std::min` и `std::max` в стандартной библиотеке есть безопасные версии, принимающие аргументы по значению и также возвращающие результат по значению. Более того, они "поддерживают" более двух аргументов.

```
const int x = 11;
const int y = 20;
auto&& y = std::min({x, 10, 15, y}); // OK
```

Может показаться, что проблема возврата ссылок касается только `const`-ссылок. С неконстантными ссылками никаких паразитных продлений жизни нет, и всё должно быть хорошо. Однако это не совсем так.

Все вышеописанное рассматривало только свободные функции и, что то же самое, статические методы классов.

Но с методами классов возврат ссылок — обычное дело. И проблемы с ними те же, но менее явные. Неявность связана с передачей указателя `this` на текущий объект.

Так, например, безопасная реализация условного `Builder` с поддержкой вызовов методов по цепочке оказывается весьма нетривиальной.

```
class VectorBuilder {
  std::vector<int> v;

public:
  VectorBuilder& Append(int x) {
    v.push_back(x);
    return *this;
  }

  const std::vector<int>& GetVector() { return v; }
};
```

```
int main() {
  auto&& v = VectorBuilder{}
              .Append(1)
              .Append(2)
              .Append(3)
              .GetVector(); // висячая ссылка
}
```

Проблема опять в умирающем объекте, вернувшем ссылку на свое содержимое.

Если мы перегрузим лишь `GetVector`, чтобы различать *`lvalue`* и *`rvalue`*, проблема не исчезнет:

```
class VectorBuilder {
  ...
  const std::vector<int>& GetVector() & {
    std::cout << "As Lvalue\n";
    return v;
  }

  std::vector<int> GetVector() && {
    std::cout << "As Rvalue\n";
    return std::move(v);
  }
};
```

Мы получим сообщение "As Lvalue". Цепочка `Append` неявно превратила безымянный временный объект в не совсем временный.

`Append` также нужно перегрузить для разбора случаев `rvalue` и `lvalue`:

```
class VectorBuilder {
  ...
  // lvalue
  VectorBuilder& Append(int x) & {
    v.push_back(x);
    return *this;
  }

  VectorBuilder&& Append(int x) && {
    v.push_back(x);
    return std::move(*this);
  }
};
```

Мы справились с висячей ссылкой на содержимое вектора.

Однако если мы захотим написать так:

```
auto&& builder = VectorBuilder{}.Append(1).Append(2).Append(3);
```

опять получим висячую ссылку, но уже на сам объект `VectorBuilder`. Добавленная перегрузка `Append` тут ни при чем — неявный `this` и в исходном случае успевал прибиваться ко временному объекту и единоразово продлевать ему жизнь.

Чтобы этого избежать, нам нужно:

- либо настраивать анализатор кода, запрещающий использовать `auto&&` и `const auto&` с этим классом в правой части;
- либо жертвовать производительностью. `rvalue`-версия `Append` может всегда возвращать `VectorBuilder` по значению (с перемещением). Если `VectorBuilder` состоит из большого числа примитивных объектов, просадка производительности может быть заметной;
- либо в принципе запретить использовать `VectorBuilder` в `rvalue`-контексте:

  ```
  class VectorBuilder {
    ...
    auto Append() && = delete;
  }
  ```

  Но тогда строить цепочки, начиная с безымянного временного объекта, будет нельзя.

Также не стоит играть с цепочками операций *op*= (+=, -=, /=) над временными объектами. Для них редко обрабатывают *rvalue*-случай:

```
struct Min {
  int x;

  Min& operator += (const Min& other) {
    x = std::min(x, other.x);
    return *this;
  };
};

int main() {
  auto&& m = (Min{5} += Min {10});
  return m.x; // висячая ссылка
}
```

Программа возвращает нулевое значение. Интересно, что неопределенное поведение проявляет себя здесь тем, что компилятор сразу генерирует возврат нулевого значения. Ассемблерный код (GCC 14.1, `-std=c++20 -O3`):

```
main:
    xor eax, eax
    ret
```

Или с использованием типов стандартной библиотеки:

```
int main() {
    using namespace std::literals;
    auto&& m = "hello"s += "world";
    std::cout << m; // висячая ссылка
}
```

Эта программа, собранная GCC 10.1, `-std=c++20 -O3`, не падает, но и ничего не печатает. А если взять GCC 14.1 и те же ключи, то мы неожиданно получим на выходе "helloworld". Прямо классика неопределенного поведения.

Автовывод типов и висячие ссылки

Хоть auto, хоть не auto, всё равно ссылка висячая!

При работе со стандартными контейнерами приходится иметь дело с очень длинными и громоздкими именами типов (`std::vector<std::pair<T1, T2>>::const_iterator`).

Начиная с 11-го стандарта, ранее бестолковое[1] ключевое слово `auto` используется для указания компилятору автоматически вывести тип переменной или (начиная с 14-го стандарта) возвращаемое значение функции. Есть еще конструкция `decltype(auto)`, работающая с тем же результатом, но по иному принципу.

Не все любят автоматическое выведение типов в C++. Призывают указывать явно, т. к. код становится более понятным, особенно в случае возвращаемого значения функции:

```
template <class T>
auto find(const std::vector<T>& v, const T& x) {
  ...
  // очень длинное тело со множеством return
  ...
}
```

Что такое это `auto` в конкретном случае? `bool`? Индекс? Итератор? Ещё что-то более страшное и сложное? Лучше б явно указали...

Но выбор между "указать явно" или "использовать автовывод", на самом деле, не так прост. В обоих случаях можно наткнуться на проблемы.

Проблема явного указания типа

Длинно и много писать — решается с помощью *using*-псевдонимов. Так что это не проблема. Другое дело, что изменение типа в одном месте потребует синхронизированных изменений в других местах.

[1] Предыдущее значение слова `auto` — это storage specifier (спецификатор хранилища), указывающий локальную область видимости и существования для переменной. Но все переменные по умолчанию именно такими и являются (если не указан модификатор `static`).

И все могло быть хорошо: поменяли где-то в объявлении, получили ошибки компиляции, исправили во всех местах, на которые указали ошибки. Но в C++ есть неявное приведение типов, которое особенно жестоко наказывает при использовании ссылок.

```
std::map<std::string, int> counters = { {"hello", 5}, {"world", 5} };

// Получаем список ключей, используем string_view,
// чтобы не делать лишних копий
std::vector<std::string_view> keys;
keys.reserve(counters.size());
std::transform(std::begin(counters),
               std::end(counters),
               std::back_inserter(keys),
               [](const std::pair<std::string, int>& item) ->
                 std::string_view {
                   return item.first;
               });

// Как-то обрабатываем список ключей:
for (std::string_view k : keys) {
  std::cout << k << "\n";         // Неопределенное поведение! Висячая ссылка
}
```

Мы немного ошиблись в аргументе лямбда-функции и получили ссылку на временный объект, а вместе с ней — неопределенное поведение.

Пример проявления на конкретных опциях компилятора GCC 14.1:

- `-std=c++17 -O3`: печатает два раза "world";
- `-std=c++17`: печатает пустую строку.

Исправляем ошибку, добавляя `const` перед `string`:

```
[](const std::pair<const std::string, int>& item) -> std::string_view
```

И получаем желаемый:

```
hello
world
```

Проходят недели, код рефакторится. Словарь `counters` отъезжает в поле какого-нибудь класса. Получение и обработка ключей — его второстепенный метод. А потом внезапно выясняется, что тип значений в ассоциативном массиве надо бы поменять на меньший, например `short`.

Вы меняете его. Уже не помните про метод обработки ключей. Компилятор не ругается.

Запускаете тестирование и возвращаетесь к той же самой ошибке.

```
int main() {
  std::map<std::string, short> counters = { {"hello", 5}, {"world", 5} };
```

```
  // получаем список ключей, используем string_view,
  // чтобы не делать лишних копий
  std::vector<std::string_view> keys;
  keys.reserve(counters.size());
  std::transform(std::begin(counters),
                 std::end(counters),
                 std::back_inserter(keys),
                 [](const std::pair<const std::string, int>& item) ->
                   std::string_view {
                     return item.first;
                 });
  // как-то обрабатываем список ключей:
  for (std::string_view k : keys) {
    std::cout << k << "\n"; // Неопределенное поведение! Висячая ссылка!
  }
}
```

Вывод программы, собранной компилятором GCC 14.1:

- с ключами `-std=c++17 -O3`: опять два раза "world";
- с ключами `-std=c++17`: печатает пустую строку.

В этом примере решением будет замена явного типа аргумента лямбда-функции на `const auto&`.

Другой пример, но уже не с аргументом, а с возвращаемым значением:

```
template <class K, class V>
class MapWrapper : private std::map<K, V> {
public:
  template <class Predicate>
  const std::pair<K, V>& find_if_or_throw(Predicate p) const {
    auto item = std::find_if(this->cbegin(), this->cend(), p);
    if (item == this->cend()) {
      throw std::runtime_error("not found");
    }
    return *item;
  }
}
```

Опять ошиблись. Опять надо исправлять. `std::map` может в будущем поменяться на что-то другое, где итератор возвращает уже не настоящий `pair`, а прокси-объект. Универсальным решением будет в этом случае `decltype(auto)` в качестве возвращаемого значения.

Проблемы автоматического вывода типа

Мы можем использовать автовывод как минимум в четырех различных формах.

1. "Голый" auto. Минимум проблем. В результате всегда получается тип без ссылок:

```
auto x = f(...); // но может быть не то, чего вы хотите:
                 // копия вместо ссылки


class C {
public:
   auto Name() const { return name; } // копия вместо ссылки
private:
   T name;
};
```

2. const auto& — это ссылка. А ссылка может оказаться "висячей":

```
const auto& x = std::min(1, 2);
// x - висячая ссылка
```

   При правильно настроенных предупреждениях компилятора в 90% случаев не выйдет использовать const auto& в качестве возвращаемого значения функции.

   Компилятор GCC (-std=c++17 -O3 -Werror):

```
<source>: In function 'const auto& add(int, int)':
<source>:9:14: error: returning reference to temporary
  [-Werror=return-local-addr]
    9 |     return x + y;
      |            ~~^~~
cc1plus: all warnings being treated as errors
Compiler returned: 1
```

3. auto&& — универсальная (universal или иногда forwarding[2]) ссылка. И тоже может оказаться висячей:

```
auto&& x = std::min(1, 2);
// x - висячая ссылка
```

   Возвращается значение аналогично варианту const auto&. Получаем ошибку компиляции при сборке с ключом -Werror.

4. decltype(auto) — автовывод "как объявлено". Справа — ссылка, слева — ссылка. Справа нет ссылки, слева нет ссылки. В каком-то смысле то же самое, что и auto&& при объявлении переменных, но не совсем:

```
auto&& x = 5;
static_assert(std::is_same_v<decltype(x), int&&>);
decltype(auto) y = 5;
static_assert(std::is_same_v<decltype(y), int>);
```

[2] См. **https://quuxplusone.github.io/blog/2022/02/02/look-what-they-need/**.

Разница в том, что auto&& — всегда ссылка, а decltype(auto) — то, "что объявлено в возвращаемом значении". И это может быть важно при дальнейших вычислениях с учетом типов.

decltype(auto) начинает стрелять при использовании его в качестве возвращаемого значения, требуя дополнительной внимательности при написании кода:

```
class C {
private:
  T name;
  const T const_name;
  DataT& data_ref;

public:
  decltype(auto) Name1() const {
    return name; // копия. name объявлен как T
  }

  decltype(auto) Name2() const {
    return (name); // ссылка. Выражение (name) имеет тип const T&:
                   // само по себе (name) - T&,
                   // но this помечен const, поэтому
                   // получается const T&
  }

  decltype(auto) ConstName() const {
    return const_name; // const копия. const_name объявлен как const T
  }

  decltype(auto) DataRef() const {
    return data_ref; // DataT&, как объявлено.
    // return (data_ref); будет то же самое.
    //                    const от this не распространяется дальше под
    //                    поля-ссылки и указатели.
  }

  decltype(auto) DanglingName1() const {
    auto&& local_name = Name1(); // возвращает копию.
                                 // Копия прибивается к rvalue-ссылке
    return local_name; // local_name - ссылка на локальную переменную
  }

  decltype(auto) DanglingName2() const {
    auto local_name = Name1(); // возвращает копию.
```

```
    return (local_name); // (local_name) - ссылка на локальную переменную
  }

  decltype(auto) NonDanglingName() const {
    decltype(auto) local_name = Name1(); // возвращает копию
    return local_name; // возвращает копию
  }

};
```

`decltype(auto)` — это хрупкий и тонкий механизм, способный перевернуть всё с ног на голову с помощью минимального изменения в коде: "лишних" скобок или `&&`.

Что за зверь std::string_view?

Это же `const&`, только больнее.

C++17 подарил нам тип `std::string_view`, призванный убить сразу двух зайцев:

- проблемы с перегрузками для функций, которые должны работать хорошо как с C, так и с C++ строками;
- а также программистов, периодически забывающих о правилах продления жизни временным объектам.

Итак, проблема: функция хочет считать количество вхождений какого-то символа в строку:

```
int count_char(const std::string& s, char c) {
  ...
}

count_char("hello world", 'l');
// Создастся временный объект std::string,
// выделится память, скопируется строка, а потом строка умрет, и память
// деаллоцируется. Плохо, много лишних операций
```

Так что нам нужна перегрузка для C-строк:

```
int count_char(const char* s, char c) {
  // Мы тут не знаем ничего про длину строки.
  // Она вообще null-терминированная?

  // Можем только написать код,
  // наивно рассчитывающий, что его будут вызывать правильно.
  ...
}
```

И будем либо дублировать код, немного адаптируя его под C-строки, либо сделаем функцию:

```
int count_char_impl(const char* s, size_t len, char c) {
  ...
}
```

в которую поместим весь дублирующийся код, и вызовем ее из перегрузок:

```
int count_char(const std::string& s, char c) {
  return count_char_impl(s.data(), s.size(), c);
}

int count_char(const char* s, char c) {
  return count_char_impl(s, strlen(s), c);
}
```

И тут на помощь приходит `string_view`, как раз-таки являющийся парой "указатель и размер". И убивает обе перегрузки:

```
int count_char(std::string_view s, char c) {
  ...
}
```

И всё здорово, хорошо и замечательно, кроме одного "но": `std::string_view`, по сути, является ссылочным типом, как `const&`, и его можно конструировать из временных значений. Но, в отличие от просто `const&`, никакого продления жизни не будет. Вернее, будет, но не там, где ожидается.

```
auto GetString = []() -> std::string { return "hello"; };
std::string_view sv = GetString();
std::cout << sv << "\n"; // висячая ссылка!
```

В этом примере мы, конечно, почти явно стреляем себе в ногу. Можно сделать стрельбу менее явной:

```
std::string_view common_prefix(std::string_view a, std::string_view b) {
  auto len = std::min(a.size(), b.size());
  auto common_count = [&]{
    for (size_t common_len = 0; common_len < len; ++common_len) {
        if (a[common_len] != b[common_len]) {
            return common_len;
        }
    }
    return len;
  }();
  return a.substr(0, common_count);
}
```

```
int main() {
  using namespace std::string_literals;
  {
    auto common =
      common_prefix("helloW",
                    "hello"s + "World11111111111111111111");
   std::cout << common << "\n"; // ok
  }
  {
    auto common =
      common_prefix("hello"s + "World11111111111111111111111",
                    "helloW");
    std::cout << common << "\n"; // висячая ссылка
  }
}
```

Ситуация такая же, как с ранее рассмотренным `std::min`. Только защититься от подобной функции *`common_prefix`*, обернув ее в шаблон с помощью анализа *`rvalue`*/*`lvalue`*, намного сложнее: нам нужно разобрать случаи `const char*` и `std::string` для каждого аргумента, — в общем, всё то, от чего нас введение `std::string_view` "избавило".

Влететь в `string_view` можно еще изящнее:

```
struct Person {
  std::string name;

  std::string_view Initials() const {
    if (name.length() <= 2) {
      return name;
    }
    return name.substr(0, 2); // копирование - висячая ссылка!
  }
};
```

Причем видно, что Clang (18.1.0) хотя бы выдает предупреждение, а GCC (14.1) — нет.

Clang 18.1.0 с ключом `-std=c++17 -O3`:

```
<source>:16:16: warning: returning address of local temporary object
  [-Wreturn-stack-address]
   16 |         return name.substr(0, 2); // копирование - висячая ссылка!
      |                ^~~~~~~~~~~~~~~~~
```

Всё потому, что *`std::string_view`* настолько легендарный, что в Clang сделали хоть какую-то проверку времени жизни для него.

Самоинициализация

Еще не мертв, но уже и не жив.

Область видимости объекта начинается сразу же после его объявления. В той же строчке. Поэтому в C++ очень легко сконструировать синтаксически корректное выражение, использующее еще не сконструированный объект.

```
// просто и прямолинейно
int x = x + 5; // неопределенное поведение

//--------------
// менее явно
const int max_v = 10;

void fun(int y) {
 const int max_v = [&]{
   // локальный max_v перекрывает глобальный max_v
   return std::min(max_v, y);
 }();
 ...
}
```

Конечно, такой код вряд ли кто-то будет писать целенаправленно. Но он может возникать самопроизвольно при применении средств автоматического рефакторинга. Локальный `max_v` во втором примере мог изначально называться как-то по-другому. Применили автоматическое переименование и получили вместо некомпилирующегося кода код с неопределенным поведением.

В следующей версии никакой проблемы не возникает:

```
const int max_v = 10;

void fun(int y) {
 const int max_v = [y]{
   // тут виден только глобальный max_v
   return std::min(max_v, y);
 }();
 ...
}
```

Код, уходящий в область неопределенного поведения при добавлении лишь одного символа. Всё как мы любим.

Такой код синтаксически валиден, и никто не собирается его запрещать. Более того, он еще и не всегда приводит к UB. Грубо говоря, к UB приводит только использование с разыменованием ссылки на этот объект. Почему грубо? Потому что пра-

вила такие же, как и с разыменованием nullptr, т. е. довольно путанные, а не просто "никогда нельзя — всегда UB". Хотя использование такой радикальной трактовки убережет вас от многих бед[3].

Мы запросто можем сослаться на еще неинициализированный объект и вызвать на нем какой-нибудь статический метод. Например, если имя класса слишком длинное и нам лень его писать, или 80 символов классического терминала не хватает, чтобы его вместить.

```
struct ExtremelyLongClassName {

  using UnspeekableInternalType = size_t;

  UnspeekableInternalType val;

  static UnspeekableInternalType Default() { return 5;}
};

// вместо ExtremelyLongClassName::Default()
ExtremelyLongClassName x { x.Default() + 5 }; // Ok, хорошо определено
```

Или, например, в неисполняемом decltype контексте: у типов слишком длинные имена. Не проблема, спросим их у самих себя!

```
ExtremelyLongClassName z {
  [] ()-> decltype(z.Default()) { // Ok, хорошо определено
    // сложные вычисления
    return 1;
  }()
};
// Код выше эквивалентен более многословному:
ExtremelyLongClassName y {
  [] ()-> ExtremelyLongClassName::UnspeekableInternalType {
    // сложные вычисления
    return 1;
  }()
};
```

Эти примеры прекрасно компилируются и совершенно корректны.

Также возможность ссылаться на переменную в процессе ее инициализации может оказаться полезна в каких-то специфических случаях, в которых вам зачем-то нужен объект, внезапно ссылающийся сам на себя. А такие случаи хоть и специфичны, но совсем не редки! Self-referential objects — широко встречающийся паттерн:

[3] Кстати, разыменование нулевого указателя — это, можно сказать, прямо-таки философская тема. Если не верите в глубину вопроса, предлагаю на досуге посмотреть весьма интересный доклад JF Bastien "*(char*)0 = 0; - What Does the C++ Programmer Intend With This Code?" (**https://clck.ru/3Gfu7v**).

кольцевые списки, графические интерфейсы с вложенными виджетами, уведомляющими друг друга, и многое другое.

```
struct Iface {
  virtual ~Iface() = default;
  virtual int method(int) const = 0;
};

struct Impl : Iface {
  explicit Impl(const Iface* other_ = nullptr) : other(other_) {
  };

  int method(int x) const override {
    if (x == 0) {
      return 1;
    }
    if (other){
      return x * other->method(x - 1);
    }
    return 0;
  }
  const Iface* other = nullptr;
};

int main() {
  Impl impl {&impl};
  std::cout << impl.method(5);
}
```

Точно таким же образом, но более запутанно, можно завязать объекты в узел, используя делегирующие конструкторы:

```
Impl() : Impl(this) {}
```

или списки инициализации

```
struct Impl : Iface {
    const Iface* other = this;
};
```

Избежать случайного использования объекта при инициализации его же самого можно, следуя правилу *AAA* (almost always auto):

> "Всегда, если это возможно, использовать запись `auto x = T {...}` для объявления и инициализации переменных".

В такой записи использование объявляемой переменной внутри инициализирующего выражения дает ошибку компиляции. Например, для:

```
auto x = x + 1;
```

компиляторы дружно скажут:

- GCC: error: use of 'x' before deduction of 'auto';
- MSVC: error C3536: 'x': cannot be used before it is initialized;
- Clang: error: variable 'x' declared with deduced type 'auto' cannot appear in its own initializer.

std::vector и инвалидация ссылок

В стандартной библиотеке C++ не очень много последовательных контейнеров с динамической длиной:

- `std::list`;
- `std::forward_list`;
- `std::deque`;
- `std::vector`.

Из них `std::vector` используется в большинстве случаев, а остальные — только если их особенности становятся действительно необходимыми и дают заметную разницу в улучшении производительности. Так, например, возможность вставки в произвольную позицию за константное число операций в `std::list` не дает преимущества в сравнении с `std::vector` (требует линейного времени), пока контейнеры недостаточно большие или размер элементов мал.

`std::vector`, будучи самым эффективным контейнером, является еще и самым небезопасным из-за инвалидации ссылок и итераторов.

Неосторожное использование `std::vector` вкупе с обилием засахаренных синтаксических конструкций очень легко приводит к неопределенному поведению.

Простенький пример с очередью задач:

```
std::optional<Action> evaluate(const Action&);

void run_actions(std::vector<Action> actions) {
  for (auto&& act: actions) { // Неопределенное поведение
    if (auto new_act = evaluate(act)) {
      actions.push_back(std::move(*new_act)); // Неопределенное поведение
    }
  }
}
```

Красиво, коротко, с неопределенным поведением и неправильно:

- push_back может вызвать реаллокацию вектора. Итераторы begin/end инвалидируются — цикл продолжится по уничтоженным данным;
- если реаллокации не произойдет, цикл пройдет только по тому набору элементов, что были в векторе изначально. До добавленных в процессе дело не дойдет.

Корректный код:

```
void run_actions(std::vector<Action> actions) {
  for (size_t idx = 0; idx < actions.size(); ++idx) {
    const auto& act = actions[idx];
    if (auto new_act = evaluate(act)) {
      actions.push_back(std::move(*new_act));
    }
  }
}
```

В какой-то момент нам захотелось по-быстрому добавить логирование, чтобы что-то проверить:

```
void run_actions(std::vector<Action> actions) {
  for (size_t idx = 0; idx < actions.size(); ++idx) {
    const auto& act = actions[idx];
    if (auto new_act = evaluate(act)) {
      actions.push_back(std::move(*new_act));
    }
    std::cerr << act.Id() << "\n"; // Неопределенное поведение!
  }
}
```

И у нас опять неопределенное поведение: *push_back* может вызвать реаллокацию вектора, и тогда ссылка *act* станет висячей.

Корректный код:

```
void run_actions(std::vector<Action> actions) {
  for (size_t idx = 0; idx < actions.size(); ++idx) {
    if (auto new_act = evaluate(actions[idx])) {
      actions.push_back(std::move(*new_act));
    }
    std::cerr << actions[idx].Id() << "\n";
  }
}
```

Этот простой паттерн с инвалидацией ссылок в векторе может очень легко спрятаться под слоем абстракций. Например, цикл обработки является публичным методом класса TaskQueue, а обработка одной задачи — его приватный метод. В таком случае изменение в одном методе, совершенно корректное в рамках него, приведет к UB из-за неявного влияния на другой метод.

Кое-как защититься от подобной неприятности можно с помощью статических анализаторов, работающих с потоком исполнения программы. Например, в PVS-Studio есть диагностика V789, выявляющая именно такие случаи. Также проблема точно ловится санитайзерами или утилитами проверки памяти (например, valgrind). Если, конечно, у вас достаточно хорошие тесты.

В языке Rust проблема ловится на этапе компиляции с помощью borrow checker'а.

Если вы можете позволить себе просадку производительности, лучше использовать специализированные контейнеры (или адаптеры контейнеров) для специфичных задач.

Например, адаптеры `std::queue` и `std::stack` по умолчанию используют контейнер `std::deque`, который не инвалидирует ссылки при добавлении новых элементов. А также ни `std::queue`, ни `std::stack` нельзя неосторожно использовать в *`range-based-for`*: у них нет итераторов и методов `begin`/`end`.

Списки захвата лямбда-функций

C++11 подарил нам лямбда-функции и вместе с ними еще один способ неявного получения висячих ссылок.

Лямбда-функция, захватывающая что-либо по ссылке, безопасна до тех пор, пока она не возвращается куда-либо за пределы области, в которой ее создали. Как только мы куда-то возвращаем или сохраняем лямбду, начинается веселье:

```
auto make_add_n(int n) {
    return [&](int x) {
        return x + n;      // n станет висячей ссылкой!
    };
}

...
auto add5 = make_add_n(5);
std::cout << add5(5);      // Неопределенное поведение!
```

Ничего принципиально нового — тут всё те же проблемы, что и с возвратом ссылки из функции. Clang иногда способен выдать предупреждение.

Приведённый выше код, скомпилированный с помощью GCC 14.1 (`-O3 -std=c++20`), выводит значение `5`.

Если собрать с помощью Clang 18.1 (`-O3 -std=c++20`), то результатом будет `1711411576`.

И вдобавок предупреждение:

```
<source>:5:13: warning:
address of stack memory associated with parameter 'n' returned
    5 |     return [&](int x) {
      |             ^
```

```
<source>:6:20: note: implicitly captured by reference due to use here
    5 |     return [&](int x) {
      |            ~
    6 |         return x + n;
      |                    ^
```

Но стоит нам принять аргумент `make_add_n` по ссылке:

```
auto make_add_n(const int &n) {
  return [&](int x) {
    return x + n; // n станет висячей ссылкой!
  };
}
```

промолчат уже оба компилятора:

- результат при сборке GCC 14.1 (`-O3 -std=c++20`): `5`;
- результат при сборке Clang 18.1 (`-O3 -std=c++20`): `10`.

Аналогично проблему можно создать и для методов объектов:

```
struct Task {
  int id;

  std::function<void()> GetNotifier() {
    return [this]{
      // this — может стать висячей ссылкой!
      std::cout << "notify " << id << "\n";
    };
  }
};

int main() {
  auto notify = Task { 5 }.GetNotifier();
  notify(); // Неопределенное поведение!
}
```

Проявление UB:

- GCC 14.1 (`-O3 -std=c++20`): `"notify 0"`;
- Clang 18.1 (`-O3 -std=c++20`): `"notify 29863"`.

Но в этом примере можно заметить `this` в списке захвата и насторожиться. До C++20 можно отстрелить ногу чуть менее явно:

```
struct Task {
  int id;

  std::function<void()> GetNotifier() {
    return [=]{
```

```
      // this может стать висячей ссылкой!
      std::cout << "notify " << id << "\n";
    };
  }
};
```

Символ "=" предписывает захватывать всё по значению, но захватывается не поле id, а сам указатель this.

Если видите лямбду, в списке захвата которой есть this, = (до C++20) или &, обязательно проверьте, как и где эта лямбда используется. Добавьте перегрузки проверки времени жизни захватываемых переменных.

```
struct Task {
  int id;

  std::function<void()> GetNotifier() && = delete;

  std::function<void()> GetNotifier() & {
    return [this]{
      // для this теперь намного сложнее стать висячей ссылкой
      std::cout << "notify " << id << "\n";
    };
  }
};
```

Если возможно, вместо захвата по ссылке лучше использовать захват по значению или захват с инициализацией перемещением.

```
auto make_greeting(std::string msg) {
  return [message = std::move(msg)] (const std::string& name) {
    std::cout << message << name << "\n";
  };
}
...
auto greeting = make_greeting("hello, ");
greeting("world");
```

Создание кортежей

С версии C++11 в стандартной библиотеке есть замечательный шаблон класса std::tuple. Кортеж. Гетерогенный список. Отличная и полезная штука. Вот только создать кортеж, ничего не сломав и при этом получив именно то, что вы хотели, — задача совершенно нетривиальная.

Явно указывать типы элементов очень длинного контейнера — занятие не из приятных.

C++11 дал нам аж три способа сэкономить на указании типов — разные функции создания кортежей:

- `make_tuple`;
- `tie`;
- `forward_as_tuple`.

C++17 предлагает еще и возможность использовать автовывод типов и просто писать:

```
auto t = tuple { 1, "string", 1.f };
```

Всё это великолепное разнообразие дает нам возможность тонко настраивать то, какие именно типы мы хотим получить в элементах контейнера: ссылочные или нет. А также возможность ошибиться и получить проблему с lifetime.

`std::make_tuple` отбрасывает ссылки, приводит ссылки на массивы к указателям, отбрасывает `const`. В общем, применяет `std::decay_t`.

При этом есть особенный частный случай, сделанный, как обычно, из благих побуждений.

Если типом аргумента `make_tuple` является `std::reference_wrapper<T>`, то в кортеже он превращается в `T&`:

```
int x = 5;
float y = 6;
auto t = std::make_tuple(std::ref(x),
                         std::cref(y),
                         "hello");
static_assert(std::is_same_v<decltype(t), // компилируется
              std::tuple<int&,
                         const float&,
                         const char*>>);
```

Конструктор с автовыводом типов особый случай `std::reference_wrapper` не рассматривает. Но decay происходит. Это тоже успешно компилируется:

```
int x = 5;
float y = 6;
auto t = std::tuple(std::ref(x), std::cref(y), "hello");
static_assert(std::is_same_v<decltype(t),
                             std::tuple<std::reference_wrapper<int>,
                                        std::reference_wrapper<const float>,
                                        const char*>>);
```

`std::forward_as_tuple` всегда конструирует кортеж ссылок. И, соответственно, можно получить ссылку на мертвый временный объект:

```
int x = 5;
auto t = std::forward_as_tuple(x, 6.f, std::move("hello"));
static_assert(
```

```
  std::is_same_v<
    decltype(t),
    std::tuple<int&,
               float&&,
               const char (&&) [6]>>); // Да, это rvalue-ссылка на массив
std::get<1>(t); // Неопределенное поведение!
```

`std::tie` конструирует кортеж только из lvalue-ссылок. И подорваться на нем сложнее, но всё равно можно, если вы захотите полученный кортеж возвращать из функции. Но эта ситуация совершенно аналогична случаям возврата любых ссылок из функций:

```
template <class... T>
auto tie_consts(const T&... args) {
  return std::tie(args...);
}

int main(int argc, char **argv) {
  auto t = tie_consts(1, 1.f, "hello");
  static_assert(std::is_same_v<decltype(t),
                               std::tuple<const int&,
                                          const float&,
                                          const char (&)[6]>>);
  std::cout << std::get<1>(t) << "\n"; // Неопределенное поведение
}
```

Общие рекомендации:

1. Для создания возвращаемых кортежей использовать `make_tuple` с явным указанием `cref`/`ref` либо конструктор, если ссылки не нужны.
2. `std::tie` использовать только для того, чтобы временно представить набор переменных в виде кортежа:

   ```
   std::tie(it, inserted) = map.insert({x, y});  // распаковка кортежей
   std::tie(x1, y1, z1) == std::tie(x2, y2, z2); // покомпонентное сравнение
   ```

3. `std::forward_as_tuple` использовать лишь при передаче аргументов. Нигде не сохранять получаемый кортеж.

И в конце бонус.

Особые любители Python могут захотеть попробовать использовать `std::tie` для выполнения обмена значений переменных:

```
// x, y = y, x
int x = 5;
int y = 3;
std::tie(x, y) = std::tie(y, x);
std::cout << x <<  " " << y;
```

У нас тут не Python, поэтому поведение этого кода не определено. Но не печальтесь. Всего лишь unspecified. В результате вы получите либо `5 5`, либо `3 3`.

Внезапная мутабельность

Был теплый весенний денёк. Попивая чай, я медленно и лениво пролистывал студенческие работы. Я бы мог сказать, что ничего не предвещало беды, но, увы, работы были выполнены на C++.

Внезапно глаз зацепился за безобидную строчку, используемую для диагностического логирования:

```
printf("from %s -- to %s",
       storage[from].value.c_str(), storage[to].value.c_str());
```

Ничего страшного в ней нет, верно? Но в тот момент меня охватил ужас. И сейчас я поделюсь этим ужасом с вами.

В этой строке спрятана невероятная возможность для багов, неожиданных падений и неопределенного поведения!

Любая C++ строка кода очень сильно зависит от контекста, в котором расположена. Что мы можем предположить, глядя на один лишь этот `printf`?

- `storage` — это какого-то толка ассоциативный контейнер.
- В `storage` хранятся элементы, имеющие, по-видимому, строковое поле `value`. Очень вероятно, что поле имеет тип `std::string`.
- Написавший этот `printf`, вероятно, полагает, что оба ключа `from` и `to` в контейнере присутствуют.

Ладно. Теперь начинается плохое: последнее предположение в любой момент может быть случайно нарушено в дальнейшей жизни кодовой базы. И при нарушении этого предположения нас ждут самые удивительные последствия! И они будут тем более удивительными, если этот `printf` спрятан под макросом и существует только при конкретных опциях компиляции, например, если максимальный уровень логирования задается во время компиляции.

Такие разные контейнеры

Если `from` или `to` нет в списке ключей `storage`, то всё зависит от того, как этот самый `storage` обрабатывает обращение к отсутствующему ключу. Для этого надо пойти и посмотреть, какой тип имеет `storage`:

- это массив или вектор — привет, `array overrun` и неопределенное поведение;
- это `std::map` или `std::unordered_map` — вам сегодня повезло, у вас вызвался default-конструктор и получились пустые строчки. Хотя это, скорее всего, не то, что вы хотели, и новосозданный элемент вам где-нибудь да навредит.

Всё? Ничего не забыли?

Мы забыли, что стандартными контейнерами из STL дело не ограничивается. Контейнеры могут быть и из других библиотек. И в случае с ассоциативными контейнерами такое встречается крайне часто. Класс `std::unordered_map` в силу требования стандарта к стабильности ссылок на элементы и гарантий итерирования не может быть реализован эффективно. Он недружелюбен к кешу и проигрывает в бенчмарках почти всегда. Поэтому в реальных приложениях часто используются альтернативные реализации, пренебрегающие теми или иными гарантиями.

Один из популярных вариантов — семейство "плоских" хеш-таблиц с открытой адресацией. Все элементы хранятся в одном непрерывном участке памяти. Нетрудно догадаться, что если новому элементу в этом участке не найдётся места, то для его вставки придётся память реаллоцировать. И ни о какой стабильности ссылок на элементы речи быть не может.

А теперь снова возвращаемся к нашей строке:

```
printf("from %s -- to %s",
       storage[from].value.c_str(), storage[to].value.c_str());
```

Если `storage` — это хеш-таблица, имеющая схожие со стандартной поведение и интерфейс (например, `abseil::flat_hash_map`), обращение через `operator[]` модифицирует контейнер, и нас ждут разные варианты в зависимости от заполненности таблицы и наличия ключей. Но всех их нам нужно свести к одному вопросу: при обращении к какому ключу произойдет реаллокация таблицы?

Однако не спешите размышлять про `from` и `to`, ведь порядок вычисления аргументов функции не специфицирован! Обращение к ключам может быть в ЛЮБОМ порядке! А это лишь добавит остроты расследованию бага, если вы столкнётесь с ним в работе.

Но я позволю себе считать, что в нашем случае сначала идёт обращение к `from`, а затем уже к `to`.

Вариант отсутствия обоих ключей в принципе эквивалентен варианту отсутствия только ключа `to`, поэтому им и ограничимся.

```
auto& from_value = storage[from].value; // (1)
auto& to_value = storage[to].value      // (2)
```

(1) — вернет ссылку на поле в существующем или новосозданном элементе, всё нормально. Даже если мы возьмем `c_str()`, тоже ничего страшного не произойдет. Контейнер управляет памятью, висячих указателей нет.

(2) — если `to` отсутствует, то либо контейнер реаллоцируется, либо нет. Если контейнер не реаллоцируется, баг останется незамеченным. Иначе ссылка *`from_value`* инвалидируется!

Короткие строки и длинные баги

Победа? Мы разобрали баг до конца?

На самом деле нет. Ведь выше сознательно был опущен вызов `c_str()`, присутствовавший в оригинальной строке. Благодаря ему баг мог оставаться незамеченным и не валить ваши тесты! Всё дело в SSO (small string optimization).

Если `storage[from].value` имеет тип `std::string`, то на большинстве современных реализаций страшное падение произошло бы только при использовании коротких строк!

Упрощенно `std::string` выглядит как:

```
class string {
  char* data;
  size_t data_size;
  size_t capacity_size;

  const char* c_str() const { return data; }
};
```

Здесь 3 × 8 байт на 64-битной платформе. И данные строки лежат в куче. Невиданное расточительство, если строка очень короткая — 0–15 символов! Поэтому при достаточном желании и упорстве, используя `union`, можно добиться того, что для коротких строк эта структура бы воспринималась, например, так[4]:

```
class string
{
  size_t capacity;

  union
  {
    struct
    {
      char *ptr;
      size_t size;
    } heapbuf;

    char stackbuf[sizeof(heapbuf)];
  };
  const char* c_str() const {
    if (capacity > sizeof(heapbuf))
      return heapbuf.ptr;
    else
      return stackbuf;
  }
};
```

[4] Есть очень много разных вариантов реализации SSO. Особенно с учетом терминирующего \0. О них рекомендую посмотреть доклад Николаса Ормрода "Странные детали std::string на Facebook" (Nicholas Ormrod "The strange details of std::string at Facebook") на CppCon 2016 (**https://clck.ru/3GfxVp**).

В новой реализации можно расположить строки небольшой длины внутри объекта в stackbuf, не аллоцируя буфер на куче. На основе поля capacity определяется, где хранятся символы:

- если capacity превышает размер stackbuf, то объект управляет буфером на куче и хранит символы там;
- иначе символы хранятся внутри самого объекта.

А теперь возвращаемся к извлечению строк:

```
const char* from_value = storage[from].value.c_str(); (1)
const char* to_value = storage[to].value.c_str();
```

(1) — это указатель на данные в куче или в самой структуре строки? А кто ж его знает-то!

Если from_value указывает на кучу, и контейнер эффективно использует семантику перемещения, то при реаллокации строка будет перемещена: просто скопируется указатель, и from_value останется валидным.

Иначе строка будет скопирована, и почти наверняка указатель storage[from].value.c_str() не будет равен from_value.

Хотя, конечно, есть крайне маловероятный шанс, что реаллокация реализована через realloc, и вам так чудесно повезло, что внутри *realloc* было достаточно просто передвинуть границу блока памяти.

Какие выводы из этого всего нужно сделать?

1. C++ — страшный язык. Мы видим строчку на 80 символов и, когда отладчик укажет на происходящее в ней падение, чтобы разобраться в его причинах, нужно учесть порядок вычисления аргументов, устройство контейнера, move-семантику и устройство элементов контейнера.
2. Проблема была бы куда менее чудовищной, если бы operator[] не изменял контейнер.
3. Любой рефакторинг на C++ нужно проводить крайне осторожно. Даже простую замену одной структуры данных на другую с точно таким же интерфейсом: наш баг скрыт при использовании std::map/std::unordered_map, но проявляется с другими таблицами.

Как бороться?

Мне не известны настройки статических анализаторов, которые тут помогли бы. Подобные баги может выявить только очень тщательное тестирование.

Примечание на тему статического анализа кода

Андрей Карпов: "Я выписал ряд идей для доработки PVS-Studio, чтобы выявлять ошибки описанного вида. Однако заранее понятно, что диагностика получится слабой, так как будет работать только для ограниченного набора простых случаев. Нужно точно знать значение элемента, который мы ищем, и чем наполнен контейнер. Это

очень сложная задача для статических анализаторов как с точки зрения анализа потока данных, так и вычислительных затрат на подобный анализ. Так что соглашусь с Дмитрием, что всё равно в первую очередь программисту стоит надеяться только на свою внимательность при написании кода и тестирование".

Мы можем предотвращать такие баги, изменяя подход к написанию кода. Крайне советую попрограммировать на Rust (даже если вы не будете им пользоваться в рабочем проекте), чтобы выработать привычку писать код, удовлетворяющий требованиям его borrow checker'a.

Если мы гарантируем в C++ коде, что на контейнер и данные в нем единовременно могут быть либо только const-ссылки, либо не более одной мутабельной ссылки, то ошибка станет почти невозможной. Правда, гарантировать мы этого не можем. Но можем установить ограничение, чтобы все ссылки у нас были константные:

```
const auto& const_storage = storage;

// operator[] недоступен из-за const
const auto& from_value = const_storage.at(from).value;

// operator[] недоступен из-за const
const auto& to_value = const_storage.at(to).value;

// если какой-то из ключей отсутствует, будет выброшено исключение
```

Proxy-объекты и неявные ссылки

Мы в C++ очень любим generic-код. Да и не только в C++. Чтобы всё было удобно, переиспользуемо и гибко. На то нам шаблоны и даны!

Давайте напишем немного такого generic-кода:

```
template <class T>
auto pop_last(std::vector<T>& v) {
  assert(!v.empty());
  auto last = std::move(v.back());
  v.pop_back();
  return last;
}
```

Вполне разумно завести себе подобную функцию, ведь имеющиеся pop_back у стандартных контейнеров возвращают void. Это очень неудобно на практике: чаще всего мы хотим изъять последний элемент контейнера и что-то с ним сделать, а не просто выкинуть.

Всё ли хорошо с этой функцией? Конечно, на пустом векторе будет неопределенное поведение, но мы же написали assert, так что дальше всё на откуп пользова-

телю. Пусть просто пишет корректный код, а некорректный не пишет... Еще есть вопросы к гарантиям исключений, ведь из-за них стандартный `pop_back()` ничего не возвращает. Но это тема другой главы. А в остальном вроде всё в порядке, да?

Что ж, давайте воспользуемся этой функцией!

```
std::vector<bool> v(65, true);
auto last = pop_last(v);
std::cout << last;
```

Всё хорошо? Ну вроде бы. Ничего не падает. Можно пойти разными компиляторами попроверять. Неужели никакого подвоха?

На самом деле подвох есть. Число 65 выбрано не случайно и, скорее всего (зависит от реализации), в коде неопределенное поведение, которое никак не проявляется потому, что так устроены деструкторы тривиальных типов. Но обо всём по порядку.

Паттерн Proxy и proxy-объекты

Подробно о разных паттернах проектирования мы говорить не будем. Для этого есть отдельные хорошие книги[5]. Но в общих чертах: Proxy (иногда переводят как Заместитель) — объект, который перехватывает обращения к другому объекту с тем же самым (или похожим) интерфейсом, чтобы сделать что-то. Что именно — зависит от конкретной задачи и реализации.

В стандартной библиотеке C++ есть самые разные proxy-объекты (иногда не чистые proxy, а с добавлением функционала):

- `std::reference_wrapper`;
- `std::in_ptr, std::inout_ptr` в C++23;
- `std::osyncstream` в C++20;
- арифметические операции над `valarray`[6] могут возвращать proxy-объекты;
- `std::vector<bool>::reference`.

Вот последний нам и нужен.

В стандарте C++98 приняли ужасное решение, казавшееся тогда разумным: сделать специализацию для `std::vector<bool>`. Обычно `sizeof(bool) == sizeof(char)`, но вообще для `bool` достаточно одного бита. Однако адресовать память по одному биту 99,99% всех возможных платформ не могут. Давайте для более эффективной утилизации памяти в `vector<bool>` будем паковать биты и хранить `CHAR_BIT` (обычно 8) булевых значений в одном байте (`char`).

Как итог — работать с `std::vector<bool>` нужно совершенно по-особому:

1. В нём нельзя взять адрес (указатель) на конкретный элемент.

[5] Разумеется, есть классическая "Приемы объектно-ориентированного проектирования. Паттерны проектирования" от "Банды четырех". Но советую обратить внимание и на аналоги от функционального мира.

[6] `std::valarray` — это векторы Numpy у нас дома! Со всеми любимыми перегруженными поэлементными операциями.

2. Соседние элементы налезают друг на друга.
3. `reference` — это не `bool&`.
4. При доступе к элементам используются похожие на `bool` proxy-объекты (знающие, к какому биту в байте обращаться). А значит, нужно быть аккуратным с автовыводом типов.

`reference` для `vector<bool>` выглядит примерно так:

```
class reference {
public:
  operator bool() const { return (*concrete_byte_ptr) & (1 << bitno); }
  reference& operator=(bool) {...}
  ...
private:
  uchar8_t* concrete_byte_ptr;
  uchar8_t  bitno;
}
```

В строке:

```
auto last = std::move(v.back());
```

`auto` отбрасывает ссылки, да. Но только настоящие C++ ссылки. `T&` и `T&&` превращаются в `T`. Тип `reference` в `bool` тут сам по себе никак не превратится, даже несмотря на наличие неявного `operator bool`!

И что же получается? А вот что:

```
auto pop_last(std::vector<bool>& v) {
  // v.size() == 65
  auto last = std::move(v.back());
  // last это vector<bool>::reference; != bool
  v.pop_back();
  // v.size() == 64
  // Мы полностью выкинули последний uint8/uint32/uint64
  // (зависит от реализации) из вектора.
  // last продолжает ссылаться на выброшенный элемент.
  // Если vector<bool> при выбрасывании этого элемента вызвал
  // (псевдо)деструктор, то далее при обращении через
  // last к этому элементу мы нарушаем объектную
  // модель C++, получая доступ к уничтоженному объекту -> UB.
  return last;
}
```

Но мы этого не почувствовали и не увидели при запусках, поскольку:

- `pop_back` не реаллоцирует внутренний буфер вектора;
- `~bool` ничего не делает.

Если же мы получим элемент из `pop_last()`, сохраним его, а потом сделаем с вектором еще что-то, что приведет к реаллокации буфера, UB начнет проявляться.

Больше неожиданностей!

Давайте взглянем на код:

```
int main() {
  std::vector<bool> v;
  v.push_back(false);
  std::cout << v[0] << " ";
  const auto b = v[0];
  auto c = b;
  c = true;
  std::cout << c << " " << b;
}
```

Такой код выводит `0 1 1`. Несмотря на `const`, значение `b` поменялось. Но ведь это же очевидно, да? Ведь `b` — это не ссылка, но объект, который ведет себя как ссылка!

Этот код станет еще более внезапным и интересным в C++23: если при переносе новинок в cppreference не ошиблись, нас ждет перегрузка операции присваивания через `const reference &`. И можно будет написать даже так:

```
int main() {
  std::vector<bool> v;
  v.push_back(false);
  std::cout << v[0] << "\t";  // 0
  const auto b = v[0];
  b = true;
  std::cout << v[0];          // 1
}
```

Такое поведение вполне определено, но может быть неожиданным, если вы пишете какой-нибудь универсальный шаблонный код. Опытные C++-программисты с опаской относятся к явному использованию `vector<bool>`... Но всегда ли они проверяют в шаблонной функции, принимающей `vector<T>`, что `T != bool`? Скорее всего, почти никогда (если только они не пишут публичную библиотеку).

Ну ладно, понятно с этим вектором всё. В остальных-то случаях всё хорошо же?

Конечно!

Давайте возьмем совершенно невинную функцию[7]

```
template <class T>
T sum(T a, T b)
```

[7] Этот пример прислал Сергей Шульман, GitHub — **@sgshulman**. Спасибо ему за это!

```
{
  T res;
  res = a + b;
  return res;
}
```

И случайно засунем в нее... правильно, какой-нибудь proxy-тип. (Что же это может быть?)

```
std::vector<bool> v{true, false};
std::cout << sum(v[0], v[1]) << std::endl;
```

Если нам повезет, мы получим ошибку компиляции. Так, например, в реализации MSVC у `vector<bool>::reference` нет конструктора по умолчанию. А GCC и Clang спокойненько компилируют нечто, падающее с ошибками обращения к памяти: `T res` ссылается на несуществующий вектор.

Также стоит отметить, как удивительно здесь работают неявные вызовы операторов приведения типов! Ведь на `vector<bool>::reference` не определен оператор `+`. И `return a + b;` не скомпилируется. Здесь `a` и `b` приводятся к `bool`, затем к `int`, чтобы просуммироваться и потом обратно привестись к `bool`.

Что еще нужно сделать и почему нужно быть бдительным?

`std::vector<bool>` — это просто самый известный пример объекта, порождающего proxy. Вы можете всегда написать свой класс и, если он будет эмулировать поведение тривиальных типов, устроить кому-нибудь (например, коллегам) развлечение.

Стандарт может разрешать возвращать proxy и для других типов и операций. И разработчики стандартной библиотеки могут этим воспользоваться. А могут и не воспользоваться. Так или иначе мы можем случайно или специально написать код, поведение которого будет зависеть от версии библиотеки. Например, согласно документации, операторы `*` у `std::valarray` в libstdc++ v12.1 и Visual Studio 2022 имеют разные типы возвращаемого значения.

В сторонних библиотеках также могут применяться proxy-объекты. И уж тем более в сторонних библиотеках их использование может меняться от версии к версии.

Например, proxy-объекты используются для операций над матрицами в библиотеке Eigen. Результатом произведения двух матриц оказывается не матрица, а специальный proxy-объект `Eigen::Product`. Транспонирование матрицы возвращает `Eigen::Transpose`. Да и многие другие операции порождают proxy-объекты. И если вы на одной версии написали:

```
const auto c = op(a, b);
b = d;
do_something(c);
```

и всё работало, то при обновлении всё вполне может сломаться. Вдруг *op* теперь возвращает ленивый proxy, а следующей строкой вы испортили один из аргументов?

Что делать и как бороться?

В C++ — никак. Только повышенной внимательностью. А также тщательно описывать ограничения, накладываемые на типы в шаблонах (желательно в виде концептов C++20).

Если вы разрабатываете библиотеку, то подумайте дважды, прежде чем добавить в публичный API неявные proxy. Если ну очень сильно хочется добавить, то поразмыслите, нельзя ли обойтись без неявных преобразований. Очень многие проблемы, которые мы тут рассмотрели, происходят от неявных преобразований. Может быть, лучше сделать API чуть менее удобным и более многословным, но безопасным?

Если вы пользуетесь библиотекой, то, может, лучше всё-таки указать тип переменной явно? Если вы хотите `bool` — укажите `bool`. Хотите тип элемента вектора? Укажите `vector<T>::value_type`. `auto` — это очень удобно, но только если вы знаете, что делаете.

Ситуация use-after-move

Move-семантика C++11 — важная и нужная фича, позволяющая писать более производительный код, не делающий лишних копий, аллокаций, деаллокаций, а также явно выражать намерение передачи владения ресурсом из одной функции в другую. Всё как в уже многие годы любимом на Stack Overflow языке Rust. Но по-другому.

Про move-семантику почти наверняка спрашивают на любом сколько-нибудь серьезном собеседовании. Хороший кандидат как-нибудь на пальцах да объяснит, что вот, мол, на примере вектора, один объект из другого что-то там забрать может, а эти `&&` — ну просто синтаксический костыль, потому что `const&` может принять временный объект, но под `const` потом ничего не поменяешь, а `&` принять временный объект не может, а у `by value` с конструктором копирования проблемы... В общем, так получилось. В конце концов, вы с кандидатом, может быть, напишете простенький `unique_ptr`, чтобы он точно продемонстрировал в коде, как воровать указатели из одного объекта в другой. И в теории этого должно хватать в 99% случаев.

А на практике потом встречается 1% интересного. Об этом интересном и пойдет речь далее.

Move-семантика в C++ хоть и достаточно эффективна, но всё же не до конца. Ее прилепили сверху как неплохой обходной путь, но оставили существенную проблему.

Давайте глянем на незамысловатый `unique_ptr`:

```
template<class T>
class UniquePtr {
public:
  explicit UniquePtr(T* raw) : _ptr {raw} {}
  UniquePtr() = default;
  ~UniquePtr() {
```

```
      delete _ptr;
  }
  UniquePtr(const UniquePtr&) = delete;
  UniquePtr(UniquePtr&& other) noexcept :
    _ptr { std::exchange(other._ptr, nullptr) } {}
  UniquePtr& operator=(const UniquePtr&) = delete;
  UniquePtr& operator=(UniquePtr&& other) noexcept {
    UniquePtr tmp(std::move(other));
    std::swap(this->_ptr, tmp._ptr);
    return *this;
  }
private:
  T* _ptr = nullptr;
};

...

UniquePtr<MyType> uptr = ...;
...
// что-нибудь важное с uptr
...
UniquePtr<MyType> b = std::move(uptr);
// а тут ничего не мешает сделать
// uptr = fun();
```

`std::move`, как известно, ничего не перемещает. Это просто преобразование ссылок, чтобы при вызове конструктора или оператора присваивания была выбрана нужная перегрузка с `rvalue`-ссылкой. Исходный объект, из которого произвели перемещение, никуда не девается (в отличие от Rust — там объект после перемещения становится недоступен для использования). У него когда-нибудь будет вызван деструктор. Потому мы обязаны оставить этот объект в каком-то адекватном для вызова деструктора состоянии. В нашем `UniquePtr` следует оставить `nullptr`, как это сделано в move-конструкторе.

Но что же происходит в операторе move-присваивания?

```
UniquePtr& operator=(UniquePtr&& other) noexcept {
  UniquePtr tmp(std::move(other));
  std::swap(this->_ptr, tmp._ptr);
  return *this;
}
```

Тут зачем-то используется move(copy)-and-swap... Ну как зачем: мы же, наверное, хотим грохнуть старый объект (`T`, а не указатель) и забрать владение новым? Или не хотим? Если нет, то почему бы не реализовать оператор перемещения так:

```
UniquePtr& operator=(UniquePtr&& other) noexcept {
  std::swap(this->_ptr, other._ptr);
  return *this;
}
```

Владение данными передано? Передано.

Старый объект-указатель в адекватном состоянии для вызова деструктора? Да, не хуже, чем тот, куда присваивали!

Всё отлично с точки зрения семантики перемещения C++!

Но такое поведение для `UniquePtr` как минимум неожиданное. Потому в стандартной реализации `std::unique_ptr` все-таки зануляет исходный указатель. Это же верно и для `std::shared_ptr`, `std::weak_ptr`. И это гарантируется стандартом...

И тут скрывается главная ловушка: если пустое moved-out-состояние для умных указателей гарантировано, то для других классов из стандартной библиотеки (и не только) это вообще-то не так! Совсем не так!

std::vector и другие похожие контейнеры

Поведение move-оператора перемещения для вектора описывается очень хитро и учитывает параметр, о котором вспоминают только те, кто о нем знают и заинтересованы в его настройке, — аллокатор.

В каждом экземпляре `std::vector` запрятан объект-аллокатор. Это может быть как по умолчанию (`std::allocator`) пустой объект, использующий глобальные `malloc`/`operator new`, так и что-то более специфичное. Например, вы хотите, чтобы каждый ваш вектор использовал свой уникальный предвыделенный кусок одного большого буфера, который полностью под вашим контролем.

Стандартная библиотека просит от типа-аллокатора определить свойство `propagate_on_container_move_assignment`, влияющее на то, как будет вести себя move-присваивание. Если вы пишете `A = std::move(B)`, есть три варианта:

- `propagate_on_container_move_assignment{} == true` (да, это не константа, а структура как `false_type`/`true_type`). Вектор *A* деаллоцируется, аллокатор перемещается (опять-таки с помощью move-присваивания, так что тут уж надо позаботиться о каких-то гарантиях), и содержимое забирается целиком из `B`. Вектор `B` будет пуст;
- `propagate_on_container_move_assignment{} == false`, и аллокатор в `A` и `B` один и тот же (`A.get_allocator() == B.get_allocator()`). *A* деаллоцируется, аллокатор остается на месте. Содержимое забирается из *A* в *B*;
- `propagate_on_container_move_assignment{} == false` и `A.get_allocator() != B.get_allocator()`. Вот тут начинается самое интересное: забрать ни аллокатор, ни данные целиком *A* не может. Единственный вариант — переносить каждый элемент отдельно. Но опустошать и деаллоцировать *B* не обязательно. Достаточно только перенести элементы. И в этом случае можно получить полный вектор, состоящий из moved-out-элементов.

В реализации вектора в библиотеке libc++ в третьем случае как раз-таки вектор не остается пустым. В libstdc++ же воткнут вызов `clear()`.

В этом можно убедиться на примере:

```
template <class T>
struct MyAlloc {
  using value_type = T;
  using size_type = size_t;
  using difference_type = ptrdiff_t;
  using propagate_on_container_move_assignment = std::false_type;

  T* allocate(size_t n) {
    return static_cast<T*>(malloc(n * sizeof(T)));
  }

  void deallocate(T* ptr, size_t n) {
    free(static_cast<void*>(ptr));
  }

  using is_always_equal = std::false_type;
  bool operator == (const MyAlloc&) const {
    return false;
  }
};

int main() {
  using VectorString = std::vector<std::string, MyAlloc<std::string>>;

  {
    VectorString v = {
      "hello", "world", "my"
    };
    VectorString vv = std::move(v);
    std::cout << v.size() << "\n";
    // выведет 0. Это был move-конструктор
  }

  {
    VectorString v = {
      "hello", "world", "my"
    };
```

```
    VectorString vv;
    vv = std::move(v);
    std::cout << v.size() << "\n";
    // выведет 3. Было move-присваивание
    for (auto& x : v) {
      // но каждый элемент был перемещен - тут пусто
      std::cout << x;
    }
  }
}
```

Компилируем и запускаем:

- `clang -std=c++20 -stdlib=libc++`: `0 3`;
- `clang -std=c++20`: `0 0`.

Обратите внимание, проблема только с move-присваиванием! Ну а еще это замечательный пример того, как разрыв объявления и инициализации переменной может менять поведение C++-программы!

Кстати, элементами вектора были строки. И последний цикл обращается к moved-out-строкам!

std::string

Moved-out-состояние строк также не специфицировано.

На разных ресурсах, посвященных C++, можно найти пример, выдающий неожиданный результат при компиляции старым Clang 3.7 с libc++:

```
void g(std::string v) {
  std::cout << v << std::endl;
}

void f() {
  std::string s;
  for (unsigned i = 0; i < 10; ++i) {
    s.append(1, static_cast<char>('0' + i));
    g(std::move(s));
  }
}
```

Начиная с C++11, строки в реализации тройки основных компиляторов используют SSO (small string optimization) — короткая строка хранится не в куче, а в самом объекте-строке (вместо/поверх указателей `union`). И ее копирование становится тривиальным. А тривиальные объекты (примитивы, структуры из примитивов) еще и перемещаются тривиально — простым копированием. С современными версиями GCC и Clang, с libc++, с lidstdc++ строка остается пустой после перемещения. Но полагаться на это всё же не стоит.

std::optional

Еще один забавный тип, у которого не самое ожидаемое moved-out-состояние. Если воспринимать `std::optional` как контейнер из нуля или одного элемента, то можно было бы предположить, что moved-out-состояние должно быть пустым — `nullopt`. Но это не так. Оно также не специфицировано, а значит, можно сделать кое-что интересное!

```
int main() {
    std::optional<std::string> opt1 = "looooooooooooooong";
    if (opt1) {
        std::cout << opt1->length() << "\n";
    }
    auto _ = std::move(opt1);
    if (opt1) {
        std::cout << opt1->length() << "\n";
    }
}
```

Этот код выводит два числа вместо одного

```
19
0
```

При перемещении из `std::optional` его состояние не меняется. Перемещается значение внутри него. Это позволяет `std::optional` быть тривиально перемещаемым, если хранимый тип также тривиально перемещаем, что способствует генерации более эффективного кода — просто memcpy всяко быстрее чем поэлементное копирование с изменением флага инициализации! А также позволит вам отстрелить ногу при неосторожном обращении к moved-out-значению внутри.

Что же делать?

С moved-out-состоянием объектов может быть четыре уровня гарантий.

1. **Destructor only.** Moved-out-объект годится только на то, чтоб быть уничтоженным. И больше не использоваться. Никак. Это базовая гарантия, которую вы должны обеспечить, если уж решили добавлять move-семантику к своим объектам, чтобы весь механизм автовызова деструкторов не отстрелил никому ноги.
2. **Destructor & assignment.** Теперь еще можно переиспользовать объект, присвоив ему новое значение (а потом уже пользуйтесь нормально). Объект, который можно перемещать, но нельзя потом переприсваивать, — очень редкое явление. Поэтому обычно данную гарантию объединяют с предыдущей.
3. **Valid, but unspecified.** Можно пользоваться, вызывать методы, не требующие предусловий, но что там внутри — можно только догадываться.
4. **Valid, well defined.** Всё и так ясно.

Читайте документацию, прежде чем переиспользовать незнакомый moved-out-объект! А лучше в принципе так не делать. И многие статические анализаторы способны выдать предупреждение, если у вас произошло обращение к moved-out-объекту в функции, где вы вызвали на нём `std::move`.

А ещё при реализации оператора перемещения стоит использовать `move_and_swap`-паттерн (как в `UniquePtr` в самом начале), так у вас больше шансов без больших усилий оставлять свои объекты в действительно пустом состоянии.

Циклы по диапазонам

Синтаксический сахар с ложкой дегтя.

Как мы уже выяснили ранее, константные ссылки `lvalue` (да и `rvalue`) доставляют много радости в C++ благодаря правилу продления жизни для временных объектов.

Правило хитрое и состоит не только в том, что `const&&` или `&&` продлевают жизнь временному объекту (но только первая такая ссылка). На самом деле, правило такое: все временные объекты живут до окончания выполнения всего включающего их выражения (statement). Иными словами, до ближайшей точки с запятой (`;`). Или же до окончания области видимости первой попавшейся на пути у этого временного объекта `const&`- или `&&`-ссылки, если область видимости ссылки больше, чем время жизни этого самого временного объекта.

То есть:

```
const int& x = 1 + 2; // временные объекты 1, 2,
// порождают временный объект 3 (сумма).
// Их время жизни закончится на ;
// Но мы присваиваем 3 константной ссылке.
// Ее область видимости простирается ниже, дальше ;
// Так что время жизни продлевается.
// Таким образом: 1, 2 - умирают, 3 - продолжает жить

const int& y =
  std::max([](const int& a, const int& b) -> const int&
  {
    return a > b ? a : b;
  } (1 + 2, 4), 5); // временные объекты 1, 2, 3 (сумма), 4, 5 живут до ЭТОЙ ;

// 3, 4 присваиваются константным ссылкам в аргументах лямбда-функции.
// Область видимости этих ссылок заканчивается после return
// - она МЕНЬШЕ времени жизни временного объекта.
// Ссылки ничего не продлили, но лишили временный объект будущего.
```

```
// 5 прибивается к константной ссылке в аргументе std::max.
// Со ссылками на 4, 5 успешно отрабатывает std::max -
// их время жизни еще не закончилось. Ссылки валидны.

// Ссылка-результат присваивается `y`. Продлений жизни не происходит -
// все временные объекты уже безуспешно попытали
// счастья на аргументах функций.
// Дело доходит до ; Время жизни всех объектов 1, 2, 3, 4, 5 заканчивается.
// `y` становится висячей. Занавес.
```

Вооружившись полученным пониманием, рассмотрим другой пример и перестанем опять всё понимать:

```
struct Point {
  int x;
  int y;
};

struct Shape {
public:
  using VertexList = std::vector<Point>;
  VertexList vertices;
};

Shape MakeShape() {
  return { Shape::VertexList{ {1,0}, {0,1}, {0,0}, {1,1} } };
}

int main() {
  for (auto v : MakeShape().vertices) {
    std::cout << v.x << " " << v.y << "\n";
  }
}
```

Всё работает, как и ожидается.

Собираем код с помощью (GCC 14.1, `-std=c++20 -O3`) и убеждаемся, что он печатает:

```
1 0
0 1
0 0
1 1
```

Повысим инкапсуляцию, проведем минимальный рефакторинг и сделаем *vertices* приватным полем с read-only-доступом:

```
struct Shape {
public:
```

```
  using VertexList = std::vector<Point>;
  explicit Shape(VertexList v) : vertexes(std::move(v)) {}

  const VertexList& Vertices() const {
    return vertexes;
  }

private:
  VertexList vertices;
};

...

int main() {
  for (auto v : MakeShape().Vertices()) {
    std::cout << v.x << " " << v.y << "\n";
  }
}
```

И всё сломалось. В коде неопределенное поведение.

Код, собранный стареньким GCC 10.2 (`-std=c++20 -O3`), печатает мусорные значения:

```
5741 0
243908863 -1980499632
0 0
1 1
```

А если собрать с помощью GCC 14.1 (`-std=c++20 -O3`), то дело кончается сообщением "Program terminated with signal: SIGSEGV".

Как же так? Разгадка в том, что, несмотря на то что заголовок `range-based for` выглядит как единое выражение, пишется и воспринимается как единое выражение, единым выражением он не является.

С C++17 стандарта и дальше конструкция

```
for (T v : my_cont) {
  ...
}
```

разворачивается примерно в следующее:

```
auto&& container_ = my_cont; // sic!
auto&& begin_ = std::begin(container_);
auto&& end_ = std::end(container_);
for (; begin_ != end_; ++begin_) {
  T v = *begin_;
}
```

В первом случае:

```
auto&& container_ = MakeShape().vertices;
```

временный объект Shape живет до ;. Он не встретил еще ни одной const&- или &&-ссылки. Подобъект vertices считается таким же временным. Его время жизни закончится на ;. Но он встречает &&-ссылку, область видимости которой простирается ниже и продлевает ему жизнь. Причем продлевается жизнь не только подобъекту vertices, а целиком временному объекту Shape, его содержащему.

Во втором случае:

```
auto&& container_ = MakeShape().Vertexes();
```

временный объект Shape живет до ;. Но он встречает неявную const&-ссылку в методе Vertices(). Ее область видимости ограничена телом метода. Продление жизни не происходит. Возвращается ссылка на часть временного объекта и присваивается ссылке *container_*. Дело доходит до ;. Временный Shape умирает. container_ становится висячей ссылкой. Занавес.

Вот так всё просто и сломано.

Как избежать проблемы с range-based for?

- Никогда не забывать делать rvalue-перегрузку[8] для любых const-методов.
- Никогда не использовать никакие выражения после двоеточия (:) в заголовке цикла. Только переменные или их поля.
- В C++20 использовать синтаксис range-based for с инициализатором: for (auto cont = expr; auto x : cont).
- При использовании синтаксиса с инициализатором думать, прежде чем использовать auto&& или const auto& для инициализатора. Впрочем, это не только про for...
- Использовать std::ranges::for_each.
- Не использовать range-based for в C++, пока его не починят. Но это все же последнее средство.

Стандарт C++23

Продление времени жизни объекта в заголовке range-based for было наконец-то исправлено. И теперь получить висячую ссылку стало сложнее. Однако теперь проще получить другую проблему — lifetime extension, или продление времени жизни.

Продление времени жизни объектов

Продление времени жизни временных объектов — тема обширная. И в этой серии заметок она встречалась не раз. Ведь работает эта особенность в довольно ограни-

[8] У методов классов и структур можно указывать квалификатор rvalue/lvalue. Например: VertexList Vertices() &&.

ченном числе случаев, и чаще всего можно получить висячую ссылку. Однако в данном разделе я хочу остановиться на менее очевидном случае с не совсем ожидаемыми последствиями.

В C++ при **первом** присваивании временного объекта const *lvalue*- или *rvalue*-ссылке время жизни этого объекта расширяется до времени жизни ссылки:

```
std::string get_string();
void run(const std::string&);

int main() {
  const std::string& s1 = get_string();
  run(s1); // ok, ссылка валидна
  std::string&& s2 = get_string();
  run(s2); // ok, ссылка валидна
  // но
  std::string&& s3 = std::move(get_string()); // ссылка уже
                                              // не валидна!
  // первое присваивание - ссылка в аргументе std::move,
  // ее время жизни ограничено телом move
  // аналогично для любой другой функции, принимающей и
  // возвращающей ссылку (std::move тут взят только для примера)
}
```

Чуть менее очевидная особенность: такой эффект дает ссылка не только на временный объект, но и на любой его подобъект!

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

struct User {
  std::string name;
  std::vector<int> tokens;
};

User get_user() {
  return {
    "Dmitry",
    {1,2,3,4,5}
  };
}

int main() {
  std::string&& name = get_user().name;
```

```
  // Немного грязных хаков с арифметикой указателей:
  // User жив. И мы можем добраться до его данных!
  // Соберите код с -fsanitize=address, чтобы убедиться!
  Auto& v = *(std::vector<int>*)((char*)(&name) + sizeof(std::string));
  for (int x : v) {
    std::cout << x;
  }
}
```

Код, представленный выше, выведет содержимое вектора `tokens` из объекта `User`. И в этом даже нет ничего противозаконного: никаких dangling references и use-after-free. Ссылка на поле обеспечивает продление жизни всего объекта. И это может быть ссылка на сколь угодно вложенное поле:

```
struct Name {
  std::string name;
};

struct User {
  Name name;
  std::vector<int> tokens;
};

...

int main() {
  std::string&& name = get_user().name.name;
  ...
}
```

И вложенные поля даже могут быть внутри массивов! Но массивы должны быть именно в стиле старого доброго C (`T array[N]`).

```
struct Name {
  std::string name;
};

struct User {
  Name name[2];
  std::vector<int> tokens;
};

User get_user() {
  return {
    { "Dmitry", "Dmitry" },
```

```
    {1,2,3,4,5}
  };
}

int main() {
  std::string&& name = get_user().name[1].name;
  ...
}
```

Примечание про std::array

С `std::array` такой фокус не пройдет из-за перегруженного `operator[]`:

```
error: rvalue reference to type 'basic_string<...>' cannot bind to lvalue of type
'basic_string<...>'
23 |     std::string&& name = get_user().name[1].name;
```

А замена rvalue-ссылки *`std::string&& name`* на *`const std::string& name`* поможет коду скомпилироваться и упасть с ожидаемым stack-use-after-free:

```
...
struct User {
  std::array<Name, 2> name;
  std::vector<int> tokens;
};
...
int main() {
  const std::string& name = get_user().name[1].name;
  std::cout << name << "\n";
}
```

Результат запуска:

```
Program returned: 1
==1==ERROR: AddressSanitizer:
stack-use-after-scope on address0x7e6806200040 at
pc 0x5b1ce93dcf19 bp 0x7ffdc59e7770 sp 0x7ffdc59e7768
READ of size 8 at 0x7e6806200040 thread T0
```

Здорово! Но пытливый читатель уже, наверное, догадался, в чем проблема. Мы берем ссылку только на одно поле, и, наверное, собираемся работать только с ним, а объект остается жить целиком... А что, если остальные его поля держат выделенную память? А что, если нам **критически важно**, чтоб у них был вызван деструктор?

Для наглядной демонстрации проблемы я приведу пример, внезапно, не на C++, а на Rust, поскольку там необходимый для создания неприятностей тип можно взять

из стандартной коробки. Ровно как и изысканно сломанную синтаксическую конструкцию.

```
use parking_lot::Mutex;

#[derive(Default, Debug)]
struct State {
  value: u64,
}

impl State {
  fn is_even(&self) -> bool {
    self.value % 2 == 0
  }

  fn increment(&mut self) {
    self.value += 1
  }
}

fn main() {
  let s: Mutex<State> = Default::default();

  match s.lock().is_even() {
    true => {
      s.lock().increment(); // ой, двойная блокировка!
    }
    false => {
      println!("wasn't even");
    }
  }
  dbg!(&s.lock());
}
```

Этот пример уходит в deadlock: временный объект `LockGuard` в операторе `match` остается жив по совершенной нелепости[9]! А мы же вернемся к C++.

Если мы по какой-то причине решили последовать примеру Rust и сделать `mutex` явно ассоциированный с данными (как и должно быть в 95% случаев), то получим такую же проблему при неаккуратном использовании ссылок:

```
template <class T>
struct Mutex {
```

[9] Подробнее об этом можно почитать здесь: **https://fasterthanli.me** (A Rust match made in hell).

```
  T data;
  std::mutex _mutex;

  explicit Mutex(T data) : data {data} {}

  auto lock() {
    struct LockGuard {
    public:
      LockGuard(T& data,
                std::unique_lock<std::mutex>&& guard) :
        data(data), guard(std::move(guard)) {}
      std::reference_wrapper<T> data;
    private:
      std::unique_lock<std::mutex> guard;
    };

    return LockGuard(this->data, std::unique_lock{_mutex});
  }
};


int main() {
  Mutex<int> m {15};

  // двойная блокировка (взаимная блокировка, неопределенное поведение)
  // из-за LockGuard.
  // Продление времени жизни, удалим && - и все будет нормально
  auto&& data = m.lock().data;
  std::cout << data.get() << "\n";
  auto&& data2 = m.lock().data;
  std::cout << data2.get() << "\n";
}
```

"Сам себе злобный Буратино, — скажут опытные защитники C++. — Зачем ссылка, если там и так `reference_wrapper`?" И будут, разумеется, правы. Но не переживайте, в C++23 теперь есть такая же сломанная конструкция, как и `match` в Rust. И это... **`range-based-for`**!

Удивительнейшим образом изменения в стандарте, направленные на то, чтобы починить висячую ссылку в конструкции:

```
for (auto item : get_object().get_container()) { ... }
```

теперь позволяют вляпаться в точно такой же дедлок, как в Rust:

```
template <class T>
struct Mutex {
```

```
  T data;
  std::mutex _mutex;

  explicit Mutex(T data) : data {data} {}

  auto lock() {
    struct LockGuard {
    public:
      LockGuard(T& data,
                std::unique_lock<std::mutex>&& guard) :
        data(data), guard(std::move(guard)) {}
      std::reference_wrapper<T> data;

      T& get() const {
        return data.get();
      }
      private:
        std::unique_lock<std::mutex> guard;
    };

    return LockGuard(this->data, std::unique_lock{_mutex});
  }
};

struct User {
  std::vector<int> _tokens;

  std::vector<int> tokens() const {
    return this->_tokens;
  }
};

int main() {
  Mutex<User> m { { {1,2,3, 4,5} } };

  for (auto token: m.lock().get().tokens()) {
    std::cout << token << "\n";
    m.lock(); // взаимная блокировка C++23
  }
}
```

Это "исправленное" поведение реализовано лишь в новейших версиях компиляторов (GCC 15, Clang 19). Вас может ждать много удивительных открытий, когда вы их обновите!

Прямая инициализация и ссылочные поля

Инициализация в C++ — одна из самых сложных и запутанных тем среди, наверное, всех языков программирования. Ей даже отдельная книга на 400 страниц посвящена — Bartlomiej Filipek "C++ Initialization Story: A Guide Through All Initialization Options and Related C++ Areas (C++ Stories)". Так что я позволю себе не разбирать все тонкости инициализации, а затрону лишь ту прекрасную часть, которая непосредственно связана с нашими любимыми ошибками — висячими ссылками.

C++11 принес в язык списки инициализации (list/uniform initialization syntax). С фигурными скобочками. Они считаются нынче почти всегда предпочтительными. Но вот незадача: в некоторых краевых случаях их недостаточно.

Например:

```
// У вас есть структура-агрегат
struct Pair {
    int x, y;
};
// И вы хотите создать ее в куче с помощью std::make_shared/unique
auto p = std::make_unique<Pair>(1, 5);
```

В C++17 такой код не соберется. Компилятор выплюнет длинную ошибку подстановки:

```
/opt/compiler-explorer/gcc-14.2.0/include/c++/14.2.0/bits/unique_ptr.h: In
instantiation of 'std::__detail::__unique_ptr_t<_Tp> std::make_unique(_Args&& ...)
[with _Tp = Pair; _Args = {int, int}; __detail::__unique_ptr_t<_Tp> =
__detail::__unique_ptr_t<Pair>]':
<source>:9:36:   required from here
    9 |     auto p = std::make_unique<Pair>(1, 5);
      |              ~~~~~~~~~~~~~~~~~~~~~~^~~~~
/opt/compiler-explorer/gcc-14.2.0/include/c++/14.2.0/bits/unique_ptr.h:1076:30: error:
new initializer expression list treated as compound expression [-fpermissive]
 1076 |     { return unique_ptr<_Tp>(new _Tp(std::forward<_Args>(__args)...)); }
      |                              ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/opt/compiler-explorer/gcc-14.2.0/include/c++/14.2.0/bits/unique_ptr.h:1076:30: error:
no matching function for call to 'Pair::Pair(int)'
<source>:4:8: note: candidate: 'Pair::Pair()'
```

В C++20 решили это недоразумение исправить, так что теперь у нас есть еще один uniform, то есть direct, initialization syntax. С круглыми скобочками.

```
auto p = std::make_unique<Pair>(1, 5); // теперь компилируется в C++20
```

Ну добавили и добавили... Стало же удобнее? Можно теперь всегда круглые скобки использовать?.. Не спешите. C++ не был бы самим собой, если бы с новой фичей не поставлялся бы новый подвох.

Если в вашей структуре есть ссылочные const&- или &&-поля, то с помощью list initialization syntax вы можете спокойно инициализировать их временными значениями. Время жизни временных объектов будет продлено. А вот в случае direct initialization syntax — нет. Вы получите висячую ссылку. И никакой диагностики от компилятора стандарт не требует.

```
struct S {
    const int& x;
    const std::string& s;
};

// списочная инициализация
int main() {
    S s { 1 + 1,
         "hellooooooooooooooooooooooo0000000" };
    return s.s.length(); // все отлично. Возвращает 34
}

// прямая инициализация
int main() {
    S s ( 1 + 1, "hellooooooooooooooooooooooo0000000" );
    return s.s.length(); // Бум! Неопределенное поведение.
    // Возвращает что угодно. GCC14 -std=c++23 -O3. Возвращает 98
}
```

Просто не пишите такой код, делов-то!

Но похожий код легко может случайно родиться из шаблонов:

```
template <class T1>
struct Wrapper {
    T1 first;
};

auto make(auto f) {
    using Result = Wrapper<decltype(f())>;
    // Ok, пока f не возвращает ссылки
    return std::make_unique<Result>(f() + 10);
}
```

Или при неосторожном рефакторинге с оптимизациями:

```
struct Config { ... }; // Большой объект
struct Widget {
```

```
    // было
    // Config config;
    // Вы обнаружили, что копируете один и тот же Config сотни раз.
    // И решили расшарить его между виджетами по const-ссылке.
    const Config& config;
};
// Эта строчка после вашей оптимизации продолжает молча компилироваться,
// но теперь влечет неопределенное поведение
auto parent_widget = std::make_unqiue<Widget>(read_config());
```

Нужно еще отметить, что продление жизни работает только при инициализации объектов, аллоцированных на стеке. Если же вы создаете объект на куче/в собственном буфере с помощью оператора new:

```
struct S {
    const std::string& s;
};

int main() {
    auto bad = new S {  "hellooooooooooooooooooooooo0000000" };
    return  bad->s.length();
}
```

То успешно получите висячую ссылку. GCC 14 молчит. Clang 19 выдает предупреждение

```
<source>:10:25: warning: temporary bound to reference member of allocated object will
be destroyed at the end of the full-expression [-Wdangling-field]
   10 |     auto bad = new S {  "hellooooooooooooooooooooooo0000000" };
```

Значения по умолчанию для ссылочных полей

Закончить, пожалуй, нужно упоминанием еще об одном недоразумении со ссылочными полями. Им же можно задать значения по умолчанию...

```
struct Config {
    // Так можно, и это компилируется
    const std::string& s = "default value";
};
```

И теперь, если мы создадим такой объект по умолчанию, мы обнаружим, что в случае list initialization:

```
Config c1 {}; // Ok с GCC 14. Почти ok с Clang 18
c1.s.length();
/*
 >:8:15: warning: lifetime extension of temporary created by aggregate initialization
using a default member initializer is not yet supported; lifetime of temporary will
end at the end of the full-expression [-Wdangling]
```

```
    8 |     Config c {};
*/
// Но при этом работает, и санитайзеры не находят проблем
```

Но если же мы воспользуемся неявным вызовом конструктора по умолчанию:

```
// А вот так уже не все хорошо.
Config c2;
c2.s.length();


// Clang 18: <source>:3:8: error: reference member 's' binds to a temporary object
whose lifetime would be shorter than the lifetime of the constructed object
// GCC 14: компилируется. Санитайзеры обнаруживают use-after-free
// при обращении к полю s.


 =1==ERROR: AddressSanitizer: stack-use-after-scope on address 0x7448ea400088 at pc
0x000000401337 bp 0x7fff3bd6f410 sp 0x7fff3bd6f408
READ of size 8 at 0x7448ea400088 thread T0
    #0 0x401336 in std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> >::length() const /opt/compiler-explorer/gcc-
14.2.0/include/c++/14.2.0/bits/basic_string.h:1084
```

Компилятор GCC с флагом `-Wextra` выдает предупреждение:

```
<source>:3:8: warning: a temporary bound to 'Config::s' only persists until the
constructor exits [-Wextra]
```

В общем, нельзя так просто взять и проинициализировать объект со ссылочными полями и ничего себе не отстрелить.

Тернарный оператор и временные объекты

У нашей истории довольно нестандартное начало для этой книги. Заметить странности и неожиданные ловушки во встроенном операторе C++ внезапно поспособствовала новая возможность в языке Rust. Последние несколько лет я много работаю с кодовыми базами на C++ и Rust и на стыке между ними. А поскольку вполне естественно переносить ожидания из одного языка на другой, тем более когда они похожи, я решил проверить, а как же работает похожий код на C++.

Rust в версии 1.78 неожиданно расширил возможности по автоматическому продлению жизни временных объектов. Теперь в нем можно, например, написать так:

```
let uri: &str = ...;
...
let updated_uri: &str = if !query.is_empty() {
    // Можно вернуть ссылку на временную строку!
    // Ее время жизни будет автоматически продлено.
    // Ранее этот код не проходил проверку заимствований
    // и не компилировался.
```

```
    &format!("{uri}?{query}")
} else {
    uri
}
```

Раньше было сложнее совместить краткость и отсутствие лишней аллокации. Нужно либо писать явно, но довольно уродливо:

```
let uri: &str = ...;
let updated_uri_tmp: String;
let updated_uri: &str = if !query.is_empty() {
    updated_uri_tmp = format!("{uri}?{query}");
    &updated_uri_tmp
} else {
    uri
};
```

либо прибегать к специальным типам:

```
use std::borrow::Cow; // умный указатель для клонирования при записи
let updated_uri: Cow<str> = if !query.is_empty() {
    format!("{uri}?{query}").into() // обернуть в Cow::Owned
} else {
    uri.into() // обернуть в Cow::Borrowed
};
```

Теперь же особенно удобным стало использование динамически полиморфных интерфейсов:

```
let output: &mut dyn std::io::Write = match config {
    StdOut => &mut std::io::stdout(),
    File { path } => &mut std::fs::File::create(path)?,
}
```

Вернемся теперь к C++. Здесь, конечно, `switch` не такой удобный, а `if...else` не является выражением — не может возвращать значения. Но в C++ есть тернарный оператор, а вот он уже что-то возвращает.

Посмотрим на довольно распространенный сценарий: у нас есть некоторый key-value-контейнер с неизменяемой конфигурацией. Мы обращаемся к какому-то ключу, если он там есть — отлично. Иначе используем значение по умолчанию.

```
using Map = std::map<int, std::string>;
// Мы сразу рассмотрим наиболее общий оптимальный случай:
// значение по умолчанию предоставляется функцией.
// Так мы можем избежать его вычисления, если ключ есть в таблице
void test_default_getter(const Map& m, int key, auto default_getter) {
    std::cout << "try ternary ? const string& : function()\n";
    auto iter = m.find(key);
```

```
    // Используем auto и universal reference, поскольку
    // 1. Мы не знаем тип default_getter()
    // 2. Это работает
    // 3. Так рекомендуют гайдлайны -- можем еще const добавить
    auto&& value = iter != m.end() ? iter->second : default_getter();
    std::cout << "value=" << value << "\n";
    std::cout << "address=" << uintptr_t(value.data()) << "\n";
}
```

Для отладки функция выводит адрес начала содержимого строки-значения. Так мы сможем сказать, произошло ли копирование значения из хранилища или же мы успешно взяли ссылку на него — чего бы вполне хотелось во всех случаях, когда ключ в таблице присутствует.

А теперь я предлагаю вам посмотреть на следующие 7 случаев и ответить на вопрос: что именно произойдет?

```
int main() {
  const Map m {
    {42, "Value in Table"}
  };
  std::cout << "data address in map: " << uintptr_t(m.at(42).data()) << "\n";
  using namespace std::literals;
  test_default_getter(m, 42, []{  return "default"sv; });
  test_default_getter(m, 42, [s = "default"s]() -> const std::string& { return s; });
  test_default_getter(m, 42, [s = "default"s]() mutable -> std::string& { return s;
});
  test_default_getter(m, 42, [s = "default"s]() mutable -> std::string&& { return
std::move(s); });
  test_default_getter(m, 42, [s = "default"s]() -> const std::string&& { return
std::move(s); });
  test_default_getter(m, 42, []{ return "default"s; });
  test_default_getter(m, 42, []{ return "default"; });
}
```

Несмотря на все разумные ожидания, только в первых трех случаях мы действительно получим ссылку на строку в таблице. В оставшихся четырех мы неявно получим копию!

```
data address in map: 7201512
try ternary ? const string& : function()
value=Value in Table
address=7201512
try ternary ? const string& : function()
value=Value in Table
address=7201512
try ternary ? const string& : function()
```

```
value=Value in Table
address=7201512
try ternary ? const string& : function()
value=Value in Table
address=140737440180832
try ternary ? const string& : function()
value=Value in Table
address=140737440180832
try ternary ? const string& : function()
value=Value in Table
address=140737440180832
try ternary ? const string& : function()
value=Value in Table
address=140737440180832
```

У тернарного оператора в C++ совершенно удивительные правила вывода типа возвращаемого значения:

```
bool ? T&  : T&  -> T&
bool ? T&& : T&& -> T&&
bool ? T&  : T&& -> T
bool ? T&  : T   -> T
bool ? U   : T   -> U или T, что к чему приведется
```

Посмотрим на наши примеры:

```
test_default_getter(m, 42, []{  return "default"sv; });
```

`const string& : string_view` — первый тип неявно приводим ко второму. Взятие `string_view` от `string` происходит без копирования. Всё отлично... И вроде безопасно.

А что если наша таблица с конфигурацией отпимизирована хранить `string_view` на части одной большой JSON-конфигурационной строки и ключа в ней нет?

```
using RefMap = std::map<int, std::string_view>;
void test_refmap_string(const RefMap& m, int key) {
  auto iter = m.find(key);
  auto&& value = iter != m.end() ? iter->second :
std::format("value_for_key_{}_generated", key);
  std::cout << value << "\n";
}
int main() {
  const RefMap m1 {
    {42, "Value in Table"}
  };
  test_refmap_string(m1, 43);
}
```

Получаем классическую ситуацию use-after-free со string-view. Код печатает мусор:

```
D◆q◆kp◆Ki%_generated
```

Неявные приведения типов всегда очень удобны для написания некорректных программ!

Продолжаем дальше с примерами. Второй и третий:

```
  test_default_getter(m, 42, [s = "default"s]() -> const std::string& {  return s; });
  test_default_getter(m, 42, [s = "default"s]() mutable -> std::string& {  return s;
});
```

Вполне естественно работают, как ожидалось, — без копирования. В обеих ветвях тернарного оператора окажутся lvalue-ссылки, const не важен. Результатом будет lvalue-ссылка.

Четвертый и пятый

```
  test_default_getter(m, 42, [s = "default"s]() mutable -> std::string&& { return
std::move(s); });
  test_default_getter(m, 42, [s = "default"s]() -> const std::string&& { return
std::move(s); });
```

дадут `const std::string& : [const] std::string&&` в тернарном операторе и, согласно его правилам вывода, должны вернуть std::string. То есть скопировать из левого или переместить из правого. По-другому быть не может.

Условно продлевать время жизни и условно вызывать деструкторы C++, в отличие от Rust, не умеет. Если мы посмотрим снова на Rust-пример:

```
let updated_uri: &str = if !query.is_empty() {
    // Чтобы это работало в Rust, на стеке
    // уже должно быть неявно зарезервировано место под объект,
    // а также должен быть runtime-проверяемый drop-флаг,
    // означающий, что объект был инициализирован во время выполнения этой ветки!
    // Подробнее смотрите https://doc.rust-lang.org/nomicon/drop-flags.html
    &format!("{uri}?{query}")
} else {
    uri
}
```

Удивительно, C++ не позволяет себе такой неявности и сопряженных с ней накладных расходов, при этом вполне допускает неявное копирование!

Также копирование в тернарном операторе — это в некотором роде безопасное поведение по умолчанию: если результатом станет копия, то это точно не висячая ссылка!

Последние примеры. Шестой.

```
  test_default_getter(m, 42, []{  return "default"s; });
```

То же самое, что и с четвертым и пятым, только возвращается чистое временное значение, а не ссылка.

`const std::string& : std::string` даст в результате std::string. И левый аргумент всегда будет скопирован.

Последний, седьмой пример

```
test_default_getter(m, 42, []{ return "default"; });
```

Здесь же мы получаем `const std::string& : const char*`. Указатель неявно приводится к `std::string`. Значит, для правого нужно будет создавать временный объект. Условного создания временных объектов в C++ нет — копируй левый аргумент!

Ознакомившись с этим поведением, я вспомнил все те десятки и сотни раз, когда я видел или сам писал:

```
const auto& value = config.hasValue(key) ? config.GetValue(key) : "default";
```

не подозревая что я всегда делаю копию... Зато работало!

Ну хорошо, со ссылками и временными значениями понятно. Выбран некоторый условно безопасный вариант, и мы должны быть за это благодарны.

Посмотрим, как еще мы можем себе что-нибудь отстрелить. Я упоминал, что автоматическое продление времени жизни в Rust еще облегчает работу с полиморфными объектами.

```
class Base {
public:
    virtual void foo() const {
        std::cout << "base\n";
    };
};

class Derived: public Base {
public:
    void foo() const override {
        std::cout << "derived\n";
    };
};

void test_ternary_inheritance(bool cond, const Base& a) {
    const auto& x = cond ? Derived() : a;
    x.foo();
}

int main() {
    const Derived d;

    test_ternary_inheritance(true,  d);
    test_ternary_inheritance(false, d);
}
```

Поскольку мы теперь знаем, как работает тернарный оператор и что во время запуска он по условию время жизни не продляет, можно относительно легко понять, что в обоих случаях произойдет копирование из объекта a. А вместе с ним и *слайсинг* — только подобъект базового класса будет скопирован. И дважды будет выведено base.

А что если попробовать наоборот?

```
void test_ternary_inheritance_derived(bool cond, const Derived& a) {
    const auto& x = cond ? Base() : a;
    const auto& y = cond ? a : Base();
    x.foo();
    y.foo();
}

int main() {
    const Derived d;

    test_ternary_inheritance_derived(true,  d);
    test_ternary_inheritance_derived(false, d);
}
```

В свете всего того что мы уже увидели, результат окажется весьма неожиданным... Ошибка компиляции!

```
<source>: In function 'void test_ternary_inheritance_derived(bool, const Derived&)':
<source>:28:26: error: operands to '?:' have different types 'Base' and 'const
Derived'
   28 |     const auto& x = cond ? Base() : a;
      |                     ~~~~~^~~~~~~~~~~~
<source>:29:26: error: operands to '?:' have different types 'const Derived' and
'Base'
   29 |     const auto& y = cond ? a : Base();
      |
```

И это же замечательно. Если код ошибочный, а слайсинг — это чаще всего ошибка, то он не должен компилироваться. По крайней мере так происходит в GCC 14.2 и Clang 18.1. С MSVC 19 же всё молча компилируется.

Если же мы просто уберем модификатор const из параметра...

```
void test_ternary_inheritance_derived(bool cond, Derived& a) {
    const auto& x = cond ? Base() : a;
    const auto& y = cond ? a : Base();
    x.foo();
    y.foo();
}
```

То увидим, что вот оно снова, и под Clang, и под GCC, компилируется, как ожидается, со слайсингом.

Корутины и ссылки

Синтаксис async/await плотно вошел в жизнь современных разработчиков: от фронтенда до бэкенда и низкоуровневой системщины. В 2007 году он появился в F# и за следующие годы разбежался по множеству языков: C# (2012), Python (2015), JavaScript (2017), Kotlin (2018), Rust (2019), Zig (2020)[10].

Несмотря на все его неоднозначности[11], возможность писать простой линейный код вместо классического callback-спагетти для сложных асинхронных задач полезна и сокращает усилия на прототипирование.

Чтобы не отставать, C++20 также добавил долгожданную поддержку: вместо async-функций у нас относительно явные и более общие типы — корутины. И для них тоже есть await... простите, co_await! А также co_return и co_yield — в C++ одним махом решили как проблемы асинхронных-функций, так функций-генераторов... Или создали проблемы с ними...

К сожалению, поддержка есть, а вот корутин в стандартной библиотеке нет! Если хотите, реализуйте свои. Но я, пожалуй, возьму корутины из boost::asio, чтобы продемонстрировать следующий восхитительный пример.

```
#include <iostream>
#include <concepts>
#include <vector>
#include <string>
#include <ranges>
#include <chrono>

#include <boost/asio/co_spawn.hpp>
#include <boost/asio/detached.hpp>
#include <boost/asio/io_context.hpp>
#include <boost/asio/steady_timer.hpp>

using namespace boost::asio;
namespace this_coro = boost::asio::this_coro;

using namespace std::literals::string_literals;
using namespace std::chrono;

using Request = std::string;

struct MetricEmitter {
  std::string metric_class;
```

[10] В 2024 году из Zig поддержку async-синтаксиса убрали из-за проблем реализации в автономном (self-hosted) компиляторе.

[11] С необходимостью специального синтаксиса связана знаменитая проблема "цветных" функций.

```
  void emit(std::chrono::milliseconds elapsed) const {
    std::cout << metric_class << " " << elapsed << "\n";
  }
};

// Демонстрационная корутина для имитации ввода-вывода
awaitable<void> some_io(int delay) {
  steady_timer timer(co_await this_coro::executor);
  timer.expires_after(milliseconds(delay));
  co_await timer.async_wait(use_awaitable);
  co_return;
}

awaitable<void> handle_request(const Request& r) {
  co_await some_io(15);
  std::cout << "Hello " << r << "\n";
  co_return;
}

template <std::ranges::range Requests>
awaitable<void> process_requests_batch(Requests&& reqs)
requires
std::convertible_to<std::ranges::range_value_t<Requests>, Request> {
  auto executor = co_await this_coro::executor;
  // Добавляем к обработке запроса метрики времени выполнения.
  auto handle_with_metrics =
    [metrics = MetricEmitter { "batch_processor"} ](auto&& request) ->
      awaitable<void>
  {
      auto start = steady_clock::now();
      co_await handle_request(std::move(request));
      auto finish = steady_clock::now();
      metrics.emit(duration_cast<milliseconds>(finish - start));
  };
  for (auto&& r: std::move(reqs)) {
    // Запускаем конкурентное исполнение для каждого реквеста.
    co_spawn(executor, handle_with_metrics(std::move(r)), detached);
  }
  co_return;
}
```

```
awaitable<std::vector<Request>> accept_requests_batch() {
  co_return std::vector{ "Adam"s, "Helen"s, "Bob"s };
}

awaitable<void> run() {
  co_await process_requests_batch(co_await accept_requests_batch());
  co_await some_io(100);
}

int main()
{
  // Запускаем наши корутины в однопоточном контексте исполнения.
  boost::asio::io_context io_context(1);
  co_spawn(io_context, run(), detached);
  io_context.run();
}
```

Вы могли бы предположить, что этот код успешно напечатает три раза:

```
Hello <имя>
batch_processor <время обработки>
```

в каком-то порядке.

Но на самом деле он с очень большой вероятностью упадет с ошибкой сегментации.

Посмотрим, какое приветственное сообщение покажет нам address sanitizer:

```
gcc -std=c++23 -O0 -fsanitize=address
AddressSanitizer:DEADLYSIGNAL
==1==ERROR: AddressSanitizer:
SEGV on unknown address 0x00000000001b
(pc 0x7a9f129aedf4 bp 0x7a9f12a1b780 sp 0x7fff9f00a228 T0)
==1==The signal is caused by a READ memory access.
==1==Hint: address points to the zero page.
    #0 0x7a9f129aedf4  (/lib/x86_64-linux-gnu/libc.so.6+0x1aedf4)
       (BuildId: 490fef8403240c91833978d494d39e537409b92e)
    #1 0x7a9f1288b664 in _IO_file_xsputn
       (/lib/x86_64-linux-gnu/libc.so.6+0x8b664) (BuildId:
       490fef8403240c91833978d494d39e537409b92e)
    #2 0x7a9f1287ffd6 in fwrite
       (/lib/x86_64-linux-gnu/libc.so.6+0x7ffd6)
       (BuildId: 490fef8403240c91833978d494d39e537409b92e)
    #3 0x7a9f12e900ab
       (/opt/compiler-explorer/gcc-14.2.0/lib64/libasan.so.8+0x820ab)
       (BuildId: e522418529ce977df366519db3d02a8fbdfe4494)
```

```
#4 0x7a9f12ce8d1c in
   std::basic_ostream<char, std::char_traits<char> >
   & std::__ostream_insert<char, std::char_traits<char> >
   (std::basic_ostream<char,
   std::char_traits<char> >&, char const*, long)
   (/opt/compiler-explorer/gcc-14.2.0/lib64/libstdc++.so.6+0x14cd1c)
   (BuildId: 998334304023149e8c44e633d4a2c69800a2eb79)
#5 0x407e03 in handle_request /app/example.cpp:39
#6 0x40f697 in
   std::__n4861::coroutine_handle<void>::resume() const
   /opt/compiler-explorer/gcc-14.2.0/include/c++/14.2.0/coroutine:137
...
#15 0x41dcfb in
    boost::asio::detail::scheduler::run(boost::system::error_code&)
    /app/boost/include/boost/asio/detail/impl/scheduler.ipp:210
#16 0x41f27a in boost::asio::io_context::run()
    /app/boost/include/boost/asio/impl/io_context.ipp:64
#17 0x4099bf in main /app/example.cpp:75
```

Похоже, что ссылка:

```
awaitable<void> handle_request(const Request& r)
```

немножечко испортилась... Но что же пошло не так?!

Корутины — очень сложные объекты, которые обманчиво просты в использовании благодаря синтаксическому сахару. В этом же весь смысл! Поддержка `async`/`await` на уровне языка и компиляторов делает простым то, что всегда было делать сложно вручную... Так происходит в высокоуровневых и безопасных языках с автоматическим управлением памятью: Python, C#, JavaScript, Kotlin. Но не в C++. И не в Rust. (И не в Zig.)

В примере выше есть как минимум три точки отказа, содержащих ошибки. Можете подумать об этом, пока мы будем разворачивать проблемы корутин C++.

Что такое корутина?

Здесь можно поупражняться в терминологии, ведь определений много разных. Есть довольно абстрактное и высокоуровневое: корутина — это функция, которая может приостановить свою работу, и которую можно возобновить позже.

Но есть один нюанс, ведущий к частому недопониманию: "магическим" свойством на самом деле обладает не функция, а объект, который из функции возвращается.

JavaScript-разработчики знают (я надеюсь), что:

```
async function myFunction() {
  return "Hello";
}
```

то же самое что и:

```
function myFunction() {
  return Promise.resolve("Hello");
}
```

Аналогично в Rust:

```
async fn my_foo() -> String
{
  "Hello".to_string()
}
```

то же самое что и:

```
fn my_foo() -> impl Future<Output = String> {
  // создает анонимный объект Future
  async {
    "Hello".to_string()
  }
}
```

В C++ же нет специального синтаксиса для объявления функции корутинами. Вместо этого нужно явно указать тип возвращаемого значения (например, `awaitable`). И требования к этому типу специфичны и не сразу понятны.

- `awaitable` должен удовлетворять концепту `std::coroutine_handle_traits`. То есть у него должен быть ассоциированный тип: `promise = typename awaitable::promise_type`.
- Тип *`promise`* должен удовлетворять документации[12] концепта Promise — столь сложному для описания, что про корутины в C++ приходится писать отдельные книжки. Но если кратко: promise контролирует поведение операций `co_await`, `co_yield` и `co_return`.
- Инстанциация `handle = std::coroutine_handle<promise>` должна быть успешной.
- Должна быть возможность сконструировать `awaitable` с помощью `promise.get_return_object()`.

И вот только тогда внутри тела функции, возвращающей `awaitable`, можно будет (и часто нужно будет) использовать синтаксический сахар `co_await`, `co_yield`, `co_return`

```
awaitable<std::string> myFunction() {
  co_return "Hello";
}
```

который рассахаривается в нечто подобное (это приблизительный псевдокод):

```
awaitable<std::string> myFunction() {
  using Promise = awaitable::promise_type;
  using Handle = std::coroutine_handle<Promise>;
  Promise p;
  auto state = new ImplicitlyGeneratedStateMachine<Handle>(p);
```

[12] См. § 9.5.4.3 "Coroutine definitions" (**https://clck.ru/3GjVJ4**).

```
  // state = _ 0;
  // ...
  //{
  //  switch(state) {
  //     case_0: { state = _1; p.initial_suspend(); }
  //     case _1: { p.yield_value("Hello"); }
  //   }
  // }
  return p.get_return_object();
}
```

Если же ни одно из co_*-ключевых слов в теле функции не присутствует, то никаких магических преобразований не происходит! И без злого умысла, кажется, такое придумать было нельзя! Смотрите-ка!

```
awaitable<void> process_request(const std::string& r) {
  co_await some_io(1);
  std::cout << r;
}

awaitable<void> send_dummy_request() {
  return process_request("hello");
}

int main(){
  boost::asio::io_context io_context(1);
  co_spawn(io_context, send_dummy_request(), detached);
  io_context.run();
}
```

Запускаем. Проверяем. Работает? Успешно ничего не печатает... Странно... Давайте-ка уберем some_io().

```
awaitable<void> process_request(const std::string& r) {
  std::cout << r;
}
```

Запускаем. Проверяем. Получаем что? Правильно:

```
<source>: In function 'boost::asio::awaitable<void>
          process_request(const std::string&)':
<source>:19:73: warning: no return statement in function returning
 non-void [-Wreturn-type]
 19 | awaitable<void> process_request(const std::string& r) { std::cout << r; }
    |                                                                         ^
ASM generation compiler returned: 0
<source>: In function 'boost::asio::awaitable<void>
```

```
process_request(const std::string&)':
<source>:19:73: warning: no return statement in function
returning non-void [-Wreturn-type]
 19 | awaitable<void> process_request(const std::string& r) { std::cout << r; }
    |                                                                         ^
Execution build compiler returned: 0
Program returned: 132
Program terminated with signal: SIGILL
```

Ну разумеется. Ну конечно. Ведь без волшебных ключевых слов магии нет и наша process_request-функция ничего не возвращает. А это же неопределенное поведение[13]!

Добавим co_return.

```
awaitable<void> process_request(const std::string& r) {
  std::cout << r;
  co_return;
}
```

Снова пусто...

Подключаем санитайзер!

```
==1==ERROR: AddressSanitizer: stack-use-after-return on
address 0x758796100150 at pc 0x7587985d01e6
bp 0x7ffda95e0290 sp 0x7ffda95dfa50
READ of size 5 at 0x758796100150 thread T0
    #0 0x7587985d01e5
       (/opt/compiler-explorer/gcc-14.2.0/lib64/libasan.so.8+0x821e5)
       (BuildId: e522418529ce977df366519db3d02a8fbdfe4494)
    #1 0x758798428d1c in
       std::basic_ostream<char, std::char_traits<char> >&
       std::__ostream_insert<char, std::char_traits<char> >
       (std::basic_ostream<char, std::char_traits<char> >&, char const*, long)
       (/opt/compiler-explorer/gcc-14.2.0/lib64/libstdc++.so.6+0x14cd1c)
       (BuildId: 998334304023149e8c44e633d4a2c69800a2eb79)
    #2 0x405c47 in process_request /app/example.cpp:21
    #3 0x4082f5 in std::__n4861::coroutine_handle<void>::resume() const
       /opt/compiler-explorer/gcc-14.2.0/include/c++/14.2.0/coroutine:137
    #4 0x427e8b in
       boost::asio::detail::awaitable_frame_base
       <boost::asio::any_io_executor>::resume()
       /app/boost/include/boost/asio/impl/awaitable.hpp:501
```

[13] И об этом мы поговорим буквально в следующей главе.

```
    #5 0x425e2c in
       boost::asio::detail::awaitable_thread
       <boost::asio::any_io_executor>::pump()
       /app/boost/include/boost/asio/impl/awaitable.hpp:770
```

Ой, ссылка `const std::string& r` умерла, похоже. Какое несчастье. Почему?

А вот же:

```
awaitable<void> send_dummy_request() {
  return process_request("hello");
}
```

Тут тоже нет никаких магических `co_*` ключевых слов. А значит, мы просто вызываем функцию `process_request`, она возвращает объект-корутину, который...

Самое время уточнить что происходит с аргументами функций, использующих `co_*`: они неявно копируются внутрь неявно создаваемого объекта. Значение копируется как значение. Ссылка как ссылка.

```
awaitable<std::string>
myFunction(int arg1, const std::string& arg2)
{
  using Promise = awaitable::promise_type;
  using Handle = std::coroutine_handle<Promise>;
  Promise p;
  auto state = new ImplicitlyGeneratedStateMachine<Handle>(p);
  //  state = _ 0;
  //  int __arg1 { arg1 };
  //  const std::string& __arg2 { arg2 };
  //{
  //  switch(state) {
  //     case_0: { state = _1; p.initial_suspend(); }
  //     case_1: { p.yield_value("Hello"); }
  //   }
  // }
  return p.get_return_object();
}
```

То есть:

```
awaitable<void> process_request(const std::string& r) {...}

awaitable<void> send_dummy_request() {
  // Неявно конструируется локальный временный std::string,
  // ссылка на него передается в функцию process_request,
  // где ссылка дальше копируется внутрь стейт-машины.
```

```
  return process_request("hello");
  // Возвращаем стейт-машину, а локальный временный объект умирает.
  // Использование после высвобождения при попытке получить результат стейт-машины.
}
```

Нужны волшебные слова... Между тем, вы еще не чувствуете насколько всё может стать плохо в контексте шаблонов? Нет? Когда может быть совершенно не ясно, корутину нам передали или нет? Нет? Ну ничего страшного. Возьмите это в качестве упражнения на дом — написать `std::invoke`, поддерживающий корутины. А мы пока продолжим добавлять волшебные слова.

```
awaitable<void> send_dummy_request() {
  co_return process_request("hello");
}
```

```
<source>: In function
'boost::asio::awaitable<void> send_dummy_request()':
<source>:26:5: error: no member named 'return_value' in
'std::__n4861::coroutine_traits
<boost::asio::awaitable<void> >::promise_type' {aka
'boost::asio::detail::awaitable_frame
<void, boost::asio::any_io_executor>'}
   26 |   co_return process_request("hello");
      |   ^~~~~~~~
Compiler returned: 1
```

Ну естественно. Ведь `process_request` возвращает корутину... Давайте шаг за шагом напишем, что же на самом деле должно происходить. Ведь мы же хотим понять и разобраться...

```
awaitable<void> send_dummy_request() {
  auto task = process_request("hello"); // Функция вернула
                                        // корутину-стейт-машину.
  auto result = co_await task; // Нужно дождаться завершения.
  co_return result; // И вернуть результат.
}
```

Ой, небольшая заминочка...

```
<source>: In function
'boost::asio::awaitable<void> send_dummy_request()':
<source>:27:28: error: use of deleted function
'boost::asio::awaitable<T, Executor>::awaitable
(const boost::asio::awaitable<T, Executor>&)
[with T = void; Executor = boost::asio::any_io_executor]'
   27 |     auto result = co_await task;
      |                           ^~~~
```

```
In file included from /app/boost/include/boost/asio/co_spawn.hpp:22,
                 from <source>:8:
/app/boost/include/boost/asio/awaitable.hpp:123:3: note: declared here
  123 |   awaitable(const awaitable&) = delete;
      |   ^~~~~~~~
```

Это особенность boost asio. Для большей безопасности он требует применять co_await только к rvalue. Небольшое исправление.

```
auto result = co_await std::move(task);
```

И мы получаем...

```
<source>:28:10: error: deduced type 'void' for
result' is incomplete
   28 |     auto result = co_await std::move(task);
      |          ^~~~~
```

Вы еще не почувствовали, насколько всё может стать плохо с шаблонами? Ничего страшного, помним, что у C++ всегда были проблемы с таким замечательным типом void. Просто объединим co_return и co_await в одну строчку

```
awaitable<void> send_dummy_request() {
  auto task = process_request("hello"); // Функция вернула
                                        // корутину-стейт-машину.
  // Нужно дождаться завершения.
  co_return co_await std::move(task); // И вернуть результат.
}
```

Компилируется и...

```
==1==ERROR: AddressSanitizer: stack-use-after-return on address
0x76cc8d801070 at pc 0x76cc8fa541e6 bp 0x7ffed68d7510
sp 0x7ffed68d6cd0
READ of size 5 at 0x76cc8d801070 thread T0
    #0 0x76cc8fa541e5
       (/opt/compiler-explorer/gcc-14.2.0/lib64/libasan.so.8+0x821e5)
       (BuildId: e522418529ce977df366519db3d02a8fbdfe4494)
    #1 0x76cc8f8acd1c in
       std::basic_ostream<char, std::char_traits<char> >&
       std::__ostream_insert<char, std::char_traits<char> >
       (std::basic_ostream<char, std::char_traits<char> >&,
       char const*, long)
       (/opt/compiler-explorer/gcc-14.2.0/lib64/libstdc++.so.6+0x14cd1c)
       (BuildId: 998334304023149e8c44e633d4a2c69800a2eb79)
    #2 0x405c47 in process_request /app/example.cpp:20
    #3 0x408cc7 in std::__n4861::coroutine_handle<void>::resume() const
    ...
```

```
    #14 0x417be0 in boost::asio::io_context::run()
        /app/boost/include/boost/asio/impl/io_context.ipp:64
    #15 0x406b67 in main /app/example.cpp:36
```

Та же самая проблема. Строка умерла. По той же самой причине. А что если мы теперь соединим всё в одну строку кода?

```
awaitable<void> send_dummy_request() {
  co_return co_await process_request("hello");
}
```

А вот теперь всё наконец-то правильно. И печатает заветное слово "hello".

Здорово, не правда ли, как мы прошли путь от неправильного

```
awaitable<void> send_dummy_request() {
  return process_request("hello");
}
```

к правильному

```
awaitable<void> send_dummy_request() {
  co_return co_await process_request("hello");
}
```

Разве могла бы такая красота получиться, если бы C++ решил все-таки требовать маркировку [co_]async у объявления функций? Тогда бы у нас было скучно (прям как в Rust):

```
[[co_async]] awaitable<void> send_dummy_request() {
  return process_request("hello"); // Ошибка компиляции!
                                   // Несоответствие типов / co_return
                                   // не следует использовать.
}
```

Но это все была синтаксическая забава, в процессе которой мы поймали ошибку: const-ссылки, rvalue-ссылки и неявное создание временных объектов. Ссылки неявно захватываются стейт-машиной, а неявные временные объекты неявно умирают. Создавайте временные переменные явно. Контролируйте их время жизни. Избегайте ссылочных параметров у корутин при возможности. Рецепт простой. Вернемся к нашему самому первому примеру и исправим ошибки и потенциальные проблемы со ссылками.

```
// Принимаем теперь все параметры для корутин по значению.

awaitable<void> handle_request(Request r) {
  co_await some_io(15);
  std::cout << "Hello " << r << "\n";
  co_return;
}
```

```
template <std::ranges::range Requests>
awaitable<void> process_requests_batch(Requests reqs)
requires
std::convertible_to<std::ranges::range_value_t<Requests>, Request>
{
  auto executor = co_await this_coro::executor;
  // Добавляем к обработке запроса метрики времени выполнения.
  auto handle_with_metrics =
   [metrics = MetricEmitter { "batch_processor"} ](auto request) ->
     awaitable<void>
  {
      auto start = steady_clock::now();
      co_await handle_request(std::move(request));
      auto finish = steady_clock::now();
      metrics.emit(duration_cast<milliseconds>(finish - start));
  };
  for (auto&& r: std::move(reqs)) {
    // Запускаем конкурентное исполнение для каждого реквеста.
    co_spawn(executor, handle_with_metrics(std::move(r)), detached);
  }
  co_return;
}

awaitable<std::vector<Request>> accept_requests_batch() {
  co_return std::vector{ "Adam"s, "Helen"s, "Bob"s };
}

awaitable<void> run() {
  co_await process_requests_batch(co_await accept_requests_batch());
  co_await some_io(100);
}
```

Компилируем. Запускаем. И получаем... Правильно, новую ошибку сегментации!

```
=================================================================
==1==ERROR: AddressSanitizer: heap-use-after-free on address
0x511000000228 at pc 0x7c812eca71e6 bp 0x7ffded723390
sp 0x7ffded722b50
READ of size 15 at 0x511000000228 thread T0
    #0 0x7c812eca71e5
       (/opt/compiler-explorer/gcc-14.2.0/lib64/libasan.so.8+0x821e5)
       (BuildId: e522418529ce977df366519db3d02a8fbdfe4494)
```

```
    #1 0x7c812eaffd1c in
       std::basic_ostream<char, std::char_traits<char> >&
       std::__ostream_insert<char, std::char_traits<char> >
       (std::basic_ostream<char, std::char_traits<char> >&,
       char const*, long)
        (/opt/compiler-explorer/gcc-14.2.0/lib64/libstdc++.so.6+0x14cd1c)
        (BuildId: 998334304023149e8c44e633d4a2c69800a2eb79)
    #2 0x41f592 in
       MetricEmitter::emit
       (std::chrono::duration<long, std::ratio<1l, 1000l> >)
       const /app/example.cpp:24
    #3 0x40ab8e in operator() /app/example.cpp:52
    ...
    #14 0x41f354 in boost::asio::io_context::run()
       /app/boost/include/boost/asio/impl/io_context.ipp:64
    #15 0x4099ae in main /app/example.cpp:75
```

Судя по трейсу, теперь у нас умерла другая строка... Та, что была сохранена в `MetricEmitter`.

```
template <std::ranges::range Requests>
awaitable<void> process_requests_batch(Requests reqs)
requires
std::convertible_to<std::ranges::range_value_t<Requests>, Request>
{
  auto executor = co_await this_coro::executor;
  // Добавляем к обработке запроса метрики времени выполнения.
  auto handle_with_metrics =
    [metrics = MetricEmitter { "batch_processor"} ](auto request) ->
      awaitable<void>
  {
    auto start = steady_clock::now();
    co_await handle_request(std::move(request));
    auto finish = steady_clock::now();
    metrics.emit(duration_cast<milliseconds>(finish - start));
  };
  for (auto&& r: std::move(reqs)) {
    // Запускаем конкурентное исполнение для каждого реквеста.
    co_spawn(executor, handle_with_metrics(std::move(r)), detached);
  }
  co_return;
}
```

Если вы еще не догадались, позвольте напомнить еще кое о чем неявном... А также о том, что корутинами могут быть и методы классов.

```
struct MetricEmitter {
  std::string metric_class;
  void emit(std::chrono::milliseconds elapsed) const {
    std::cout << metric_class << " " << elapsed << "\n";
  }

  awaitable<void> wrap_request(Request r) const {
    auto start = steady_clock::now();
    co_await handle_request(std::move(r));
    auto finish = steady_clock::now();
    // Корутина также неявно захватывает указатель this!
    emit(duration_cast<milliseconds>(finish - start));
  }
};

// Так что, наверное, очевидно, что
auto task =
  MetricEmitter{"batch_process"}.wrap_request(request);
co_await task; // MetricEmitter умер. Будет использование после высвобождения
```

У нас в примере кое-что похожее, но другое.

```
// handle_with_metric - это анонимная структура с
// определенным operator().
auto handle_with_metrics =
  [metrics = MetricEmitter { "batch_processor"} ](auto request) ->
    awaitable<void>
{
  auto start = steady_clock::now();
  co_await handle_request(std::move(request));
  auto finish = steady_clock::now();
  // Корутина неявно захватывает this...
  // А this в этом случае - указатель на лямбда-функцию!
  metrics.emit(duration_cast<milliseconds>(finish - start));
};
// Если лямбда-функция умрет раньше, чем завершится исполнение корутины,
// будет использование после высвобождения.
```

Смотрим как мы ее вызываем

```
for (auto&& r: std::move(reqs)) {
```

```
  // Запускаем конкурентное исполнение для каждого реквеста...
  // В ФОНЕ!!! Результат вызова handle_with_metrics - корутина -
  // сохраняется где-то внутри функции boost::asio::co_spawn.
    co_spawn(executor, handle_with_metrics(std::move(r)), detached);
    // И будет обработана позже.
}
co_return; // А вот тут наша лямбда и умрет.
```

Под такие неприятные случаи у co_spawn есть перегрузка, принимающая напрямую функцию, а не awaitable<T>. Но у функции тогда не должно быть аргументов.

Ошибку можно исправить разными способами, следуя рекомендации: все параметры корутины нужно передать явно и переместить внутрь ее тела. В C++23, для методов классов, с этим может помочь deduced this.

```
struct MetricEmitter {
  std::string metric_class;
  void emit(std::chrono::milliseconds elapsed) const {
    std::cout << metric_class << " " << elapsed << "\n";
  }

  awaitable<void> wrap_request(this auto self, Request r) {
    // Self скопирован по значению!
    auto start = steady_clock::now();
    co_await handle_request(std::move(r));
    auto finish = steady_clock::now();
    // Корутина также неявно захватывает указатель this!
    self.emit(duration_cast<milliseconds>(finish - start));
  }
};
```

А из statefull[14]-лямбд возвращать корутины не рекомендуется. Убирайте список захвата!

```
template <std::ranges::range Requests>
awaitable<void> process_requests_batch(Requests reqs)
requires
std::convertible_to<std::ranges::range_value_t<Requests>, Request>
{
  auto executor = co_await this_coro::executor;
  // Добавляем к обработке запроса метрики времени выполнения.
  auto handle_with_metrics = [](auto request) -> awaitable<void> {
    auto metrics = MetricEmitter { "batch_processor"};
    auto start = steady_clock::now();
```

[14] Также их называют capturing.

```
    co_await handle_request(std::move(request));
    auto finish = steady_clock::now();
    metrics.emit(duration_cast<milliseconds>(finish - start));
  };
  for (auto&& r: std::move(reqs)) {
    // Запускаем конкурентное исполнение для каждого реквеста.
    co_spawn(executor, handle_with_metrics(std::move(r)), detached);
  }
  co_return;
}
```

Вот теперь всё работает.

```
Hello Adam
batch_processor 15ms
Hello Helen
batch_processor 15ms
Hello Bob
batch_processor 15ms
```

Отслеживать время жизни ссылок в асинхронном коде вручную крайне тяжело. Автоматика, как в Rust, делает это намного лучше, но при этом может выдавать совершенно непонятные репорты, в которых можно разобраться, только если знаешь, что именно могло пойти не так — за это `async` в Rust и не любят, и критикуют. А в качестве самого простого и продуктивного решения, чтоб ублажить статический анализатор (он же borrow checker), выбирается копирование всего подряд (`.clone()`, везде `.clone()`).

C++ отдает полный контроль вам с невероятной кучей неявных захватов ссылок! Делайте с ними что хотите и как хотите. Скомпилируется без проблем и проверок. Вы можете приложить ментальные усилия, отследить все ссылки и убедиться, что объекты не умрут не вовремя. Либо можно отчаяться, прочитать гайдлайны и передавать все и всегда по значению. Копируя и перемещая. Никаких ссылок.

Проблема со ссылками — это лишь начало. Всё становится намного хуже, если мы еще и подключим многопоточное выполнение корутин.

- Если вы захватите блокировку с помощью `std::unique_lock` и продержите ее на время исполнения `co_await`, то готовьтесь получить неопределенное поведение — `unique_lock` может быть перемещен в другой поток, не владевший блокировкой! Только внимательность, опыт и, может быть, статический анализатор[15] уберегут вас от наступания на эти грабли.
- Разумеется, еще больше возможности для состояний гонки (race conditions). Особенно если забыть что-то скопировать внутрь тела корутины.

[15] У Clang-Tidy, например, есть диагностика cppcoreguidelines-no-suspend-with-lock.

Ах да, совсем забыл: в зависимости от реализации, корутины могут быть ленивыми или не очень ленивыми (смотрите `promise::initial_suspend()`). `Boost::asio::awaitable` — ленивые. И потому мы сразу получали прекрасные ситуации use-after-free. Для корутин, у которых `promise::initial_suspend()` возвращает `suspend_never`, код вида

```
awaitable<void> process_request(const std::string& r) {
  std::cout << r;
  co_await something();
  /* r не используется */
  co_return;
}
```

способен успешно работать и долгое время не вызывать проблем.

Корутины C++ гибки, мощны и совершенно небезопасны. Надеюсь, вы настроили сборку и тесты с санитайзерами, прежде чем решили ими воспользоваться.

По состоянию на 2024 год статические анализаторы C++ частично подсвечивают подобные проблемы.

- Clang-Tidy имеет агрессивную проверку на использование ссылок в корутинах: cppcoreguidelines-avoid-reference-coroutine-parameters.
- Также может быть настроен на подсвечивание statefull лябмд: cppcoreguidelines-avoid-capturing-lambda-coroutines.
- Но они не скажут, что вам нужно использовать `co_return co_await foo()` вместо `return foo()`.

Но в общем случае это не ошибки. Можно успешно использовать ссылки и не платить за копии. Можно также избегать лишнего обертывания в слой корутины и делать `return foo()` напрямую.

ГЛАВА 3
(Не)работающий синтаксис

Забытый return

Про C и C++ иногда говорят, что это языки, в которых есть специальный синтаксис для написания невалидных программ.

В C и C++ функцией, возвращающей что-то, отличное от `void`, необязательно должен быть `return` *что-то*.

```
int add(int x, int y) {
  x + y;
}
```

Это синтаксически корректная функция, которая приведет к неопределенному поведению. Может образоваться мусор, может возникнуть провал в код следующей далее функции, а может оказаться и "всё нормально".

Красивый пример "всё нормально"

Если собрать этот код:

```
#include <stdio.h>

int f(int a, int b) {
  int c = a + b;
}

int main() {
  int x = 5, y = 6;
  printf("f(%d,%d) is %d\n", x, y, f(x,y));
  return 0;
}
```

например, с помощью GCC 5.2, то сумма великолепно "посчитается" и программа выведет:

```
f(5,6) is 11
```

Однако не стоит думать, что неопределенное поведение для такого кода сводится лишь к тому, будет распечатано корректное значение или случайное. Отсутствующий `return` вполне может приводить к падению приложения. Рассмотренный код, кстати, тоже. Достаточно поменять компилятор на GCC 14.1, и результатом выполнения программы станет:

```
Program terminated with signal: SIGILL
```

Особенную боль это недоразумение может доставить тем, кто пришел в C++ после какого-нибудь ориентированного на выражения языка, в котором похожий код абсолютно нормален:

```
fn add(x: i32, y: i32) -> i32 {
  x + y
}
```

Обоснования, согласно которым не обязательно писать в конце функции оператор `return`, следующие:

1. В функции может быть ветвление логики. В одной из веток может вызываться код, который не предполагает возврата: бесконечный цикл, исключение, `std::exit`, `std::longjmp` или что-то иное, помеченное атрибутом `[[noreturn]]`. Проверить наличие такого кода не всегда возможно.
2. Функция может содержать ассемблерную вставку со специальным кодом финализации и инструкцией `ret`.

Проверить наличие формального `return`, конечно, можно. Но нам разрешили не писать иногда (очень иногда!) чисто формальную строчку, а компиляторам разрешили не считать это ошибкой.

За несколько лет работы над большими проектами, совмещающими C++, Rust и Kotlin (аж два языка с опциональным `return`!), я видел немало забытых `return`. И сам их забывал иногда. С особым успехом `return` оказывается потерянным в пользовательских операторах присваивания. Ведь `operator =` — пожалуй, одна из немногих не-`void`-функций, результат которой почти всегда игнорируется.

С флагом `-Wreturn-type` GCC и Clang во многих случаях сообщают о проблеме. Анализаторы, например PVS-Studio, также ругаются.

Кстати, если заглянуть в коллекцию багов, собранных командой PVS-Studio, выясняется, что отсутствие `return` — это не какие-то экзотические баги, а по-прежнему очень даже распространенные. Так что проявляйте бдительность.

Единственным исключением, начиная с C++11 (или C99, если говорить про код на чистом C), является функция `main`. В ней отсутствующий `return` к неопределенному поведению не приводит и трактуется как возврат 0.

Самое раздражающее правило синтаксического анализа[1]

Помимо неопределенного поведения, в C++ есть неожиданное поведение, произрастающее из следующих фантастических возможностей языка.

Пользовательские типы и функции можно *объявлять* где попало и как попало.

```
template <class T>
struct STagged {};

using S1 = STagged<struct Tag1>;  // предобъявление структуры Tag1
using S2 = STagged<struct Tag2*>; // предобъявление структуры Tag2

void fun(struct Tag3*); // предобъявление структуры Tag3

void external_fun() {
  int internal_fun();   // предобъявление функции!
  internal_fun();
}

int internal_fun() {    // определение предобъявленой функции
  std::cout << "hello internal\n";
  return 0;
}

int main() {
  external_fun();
}
```

При этом *определять* сущности можно не везде. Типы можно определять локально — внутри функции. А функции определять нельзя.

```
void fun() {
  struct LocalS {
    int x, y;
  };                          // OK
```

[1] Most vexing parse (см. h**ttps://en.wikipedia.org/wiki/Most_vexing_parse**).

```
  void local_f() {
    std::cout << "local_f";
  }                          // ошибка компиляции
}
```

И всё могло бы быть хорошо, если бы не одно: в C++ есть конструкторы, вызов которых похож на объявление функции.

```
struct Timer {
  int val;
  explicit Timer(int v = 0) : val(v) {}
};

struct Worker {
  int time_to_work;

  explicit Worker(Timer t) : time_to_work(t.val) {}

  friend std::ostream&
  operator << (std::ostream& os, const Worker& w) {
    return os << "Time to work=" << w.time_to_work;
  }
};

int main() {

  // ЭТО НЕ ВЫЗОВ КОНСТРУКТОРА[2]!
  Worker w(Timer()); // Предъобъявление функции,
                     // которая возвращает Worker и принимает функцию,
                     // возвращающую Timer и не принимающую ничего!

  std::cout << w;    // Имя функции неявно преобразуется к указателю,
                     // который неявно преобразуется к bool.
                     // Будет выведено 1 (true)
}
```

Подобная ошибка может быть труднообнаружима, если случайно предъобъявленная функция используется в контексте приведения к `bool`, или если объект, который хотели сконструировать, сам является вызываемым (у него перегружен `operator()`).

Может показаться, что виноват именно конструктор по умолчанию класса `Timer`. На самом деле, виноват C++. В нем можно объявлять функции вот так:

```
void fun(int (val)); // Скобки вокруг имени параметра допустимы!
```

[2] Такое контринтуитивное объявление Скот Мейерс назвал *most vexing parse*.

И потому можно получать более отвратительный и труднопонимаемый вариант ошибки:

```
int main() {
  const int time_to_work = 10;
  Worker w(Timer(time_to_work)); // Предъобъявление функции,
                                 // которая возвращает Worker
                                 // и принимает параметр типа Timer.
                                 // time_to_work - имя этого параметра!

  std::cout << w;                // Печатаем 1
}
```

GCC и Clang способны предупреждать о подобном.

В C++11 и далее предлагают uniform initialization (через {}), которая не совсем универсальна и имеет свои проблемы. C++20 предлагает еще одну universal-инициализацию, но снова через ()...

Избежать проблемы можно, используя *"almost always auto"* — подход с инициализацией вида `auto w = Worker(Timer())`. Круглые или фигурные скобки здесь — это не так важно (хотя, на самом деле, важно, но в другой ситуации).

Возможно, когда-нибудь объявление функций в старом сишном стиле запретят в пользу подхода *"trailing return type"* (`auto fun(args) -> ret`). И вляпаться в рассмотренную прелесть станет значительно сложнее[3].

Неконстантные константы

В C++ есть ключевое слово `const`, позволяющее помечать значения как неизменяемые. Также в C++ есть оператор `const_cast`, позволяющий этот `const` игнорировать. И иногда за это вам ничего не будет. А иногда будет неопределенное поведение, ошибка сегментации (segfault) и прочие радости жизни.

Разница между этими "иногда" в том, что есть настоящие константы, попытка модификации которых — UB. А есть ссылки на константы, ссылающиеся не на константы. И раз на самом деле объект неконстантен, то модифицировать его можно без проблем.

Так, например, эту "фичу" можно эксплуатировать, чтобы не повторять один и тот же код для `const` и не-`const` методов класса[4]:

```
class MyMap {
public:
```

[3] Но всё равно возможно! О самых разнообразных проблемах парсинга конструкций в C++ можно посмотреть 6-минутный доклад Тимура Думле "Можно, у нас будет грамматика?" (Timur Doumler "Can I has grammar?") на конференции CppCon 2018 (**https://www.youtube.com/watch?v=tsG95Y-C14k**).

[4] И так делали до C++23. В C++23, пожалуйста, используйте deducing this.

```
  // какой-то метод с длинной реализацией:
  const int& get_for_val_or_abs_val(int val) const {
    const auto it_val = m.find(val);
    if (it_val != m.end()) {
      return it_val->second;
    }
    const auto abs_val = std::abs(val);
    const auto it_abs = m.find(abs_val);
    if (it_abs != m.end()) {
      return it_abs->second;
    }
    throw std::runtime_error("no value");
  }

  int& get_for_val_or_abs_val(int val) {
    return const_cast<int&>( // Отбрасываем const с результата.
      // Находясь в неконстантном методе,
      // мы знаем, что результат
      // в действительности не является константой,
      // и проблем не будет.
      std::as_const(*this) // Навешиваем const,
                           // чтобы вызвать const-метод,
                           // а не уйти в бесконечную рекурсию.
      .get_for_val_or_abs_val(val));
  }

  void set_val(int val, int x) {
    m[val] = x;
  }
private:
  std::map<int, int> m;
};
```

По возможности стоит избегать такого кода. Видно, что он очень хрупок — забытый или случайно удаленный `std::as_const` ломает его. И без настройки предупреждений компиляторы об этом сообщать не торопятся.

Вместо использования `const_cast` и привнесения в мир C++ еще большей нестабильности, решить проблему дублирования кода можно с помощью шаблонного метода:

```
class MyMap {
public:
  const int& get_for_val_or_abs_val(int val) const {
```

```
    return get_for_val_or_abs_val_impl(*this, val); // *this - const&
  }

  int& get_for_val_or_abs_val(int val) {
    return get_for_val_or_abs_val_impl(*this, val); // *this  - &
  }

  void set_val(int val, int x) {
    m[val] = x;
  }
private:
  template <class Self> static decltype(auto)
    get_for_val_or_abs_val_impl(Self& self, int val)
  {
    auto&& m = self.m;
    if (it_val != m.end()) {
      // Дополнительные скобки для вывода категории значения
      return (it_val->second);
    }
    const auto abs_val = std::abs(val);
    const auto it_abs = m.find(abs_val);
    if (it_abs != m.end()) {
      return (it_abs->second);
    }
    throw std::runtime_error("no value");
  }

  std::map<int, int> m;
};
```

У этого варианта есть свои недостатки, сломать его еще проще (скобки и `decltype`). Но, единожды его написав, можно рассчитывать, что странная магия отпугнет желающих этот код поправить.

Конечно, вместо `decltype(auto)` можно написать чуть больше кода с явным указанием типов возвращаемых значений.

Const и оптимизации

Операции над иммутабельными данными отлично оптимизируются, распараллеливаются и вообще ведут себя здорово.

Однако возможность заигрывать со снятием и навешиванием const где угодно в коде исключает этот ряд оптимизаций. Так, повторное обращение по константной ссылке к одному и тому же полю или методу совсем не обязано кешироваться.

Стоит отметить, что у программистов встречаются завышенные ожидания по поводу оптимизации кода компилятором, когда они добавляют побольше const. Хорошая заметка на эту тему — "Why const Doesn't Make C Code Faster" (**https://clck.ru/3HGsaJ**).

Например, итерирование по вектору не может быть оптимизировано в таком простом случае:

```
using predicate = bool (*) (int);

int count_if(const std::vector<int>& v, predicate p) {
  int res = 0;
  for (size_t i = 0; i < v.size(); ++i) // Значение v.size() нельзя
  {                                     // единожды сохранить в регистре.
    if (p(v[i])) // Конкретный p может иметь доступ к
    {            // изменяемой ссылке на этот же самый v.
      ++res;
    }
    // Код метода size() придется выполнять на каждой итерации!
  }
   return res;
}
```

Пример, запрещающий оптимизацию, может быть неочевиден, но на самом деле прост:

```
std::vector<int> global_v = {1};

bool pred(int x) {
  if (x == global_v.size()) {
    global_v.push_back(x);
    return true;
  } else {
    return false;
  }
}

int main() {
  return count_if(global_v, pred);
}
```

Этот код очень плох. Он не должен нигде встречаться. Его никто не пропустит на ревью. Но теоретически так написать можно, поэтому оптимизация не выполняется.

Если в качестве типа предиката использовать шаблонный параметр, можно привести куда более изощренные примеры без привлечения глобальных переменных.

Учитывая ограниченные возможности на автоматическую оптимизацию, подобный цикл переписывают (делая ту самую работу, которую ждали от компилятора):

```
int count_if(const std::vector<int>& v, predicate p) {
  int res = 0;

  // range-based-for не обращается к size(),
  // а один раз получает begin/end-итераторы и работает с ними.
  for (auto x : v) {
    if (p(v[i])) {
      ++res;
    }
  }
  return res;
}
```

В таком случае при передаче "нехорошего" предиката, меняющего вектор, мы получим неопределенное поведение. Но это уже совсем другая история...

Вот менее тривиальный пример const, никак не способствующего оптимизации:

```
void find_last_zero_pos(const std::vector<int>& v,
                        const int* *pointer_to_last_zero) {
  *pointer_to_last_zero = 0;
  // Опять не можем один раз сохранить значение v.size()
  for (size_t i = 0; i < v.size(); ++i) {
    if (v[i] == 0) {
      // Внутри вектора есть поля типа int* - begin, end.
      // Что, если pointer_to_last_zero указывает на один из них?!
      *pointer_to_last_zero = (v.data() + i);
    }
    // Пересчитываем size!
  }
}
```

Оставаясь в рамках рекомендуемых практик написания C++-программ, мы не можем соорудить пример, который бы демонстрировал неприменимость оптимизации. Нам мешает инкапсуляция. До приватных полей вектора мы не можем законно добраться.

Но ненормальный код не запрещен! Применим грубую силу:

```
int main() {
  std::vector<int> a = {1,2,4,0};
  const int* &data_ptr =
```

```
    reinterpret_cast<const int* &>(a); // ссылка на begin!
  find_last_zero_pos(a, &data_ptr);
}
```

И вот мы имеем парадоксальный результат: возможность написать явно некорректный код запрещает компилятору оптимизировать цикл! И вся концепция неопределенного поведения как возможности для оптимизации (ведь некорректного кода не бывает) разваливается.

Что ж, по крайней мере, для этого примера имеется некоторая стабильность: исходный цикл со счетчиком и переписанный на range-based-for закончатся на неопределенном поведении.

В современных языках (например, в Rust благодаря семантике владения) все эти циклы могут быть успешно оптимизированы.

Const, время жизни и происхождение указателей

Неизменяемые объекты всем хороши, кроме одного: это константные объекты в C++. Если они где-то засели, то их оттуда по-хорошему не выгонишь.

Что имеется в виду?

Допустим, есть структура с константным полем:

```
struct Unit {
  const int id;
  int health;
};
```

Из-за константного поля объекты `Unit` теряют операцию присваивания. Их нельзя менять местами — `std::swap` больше не работает. `std::vector<Unit>` больше нельзя просто так отсортировать... В общем, сплошное удобство.

Но самое интересное начинается, если сделать что-то такое:

```
std::vector<Unit> units;
unit.emplace_back(Unit{1, 2});
std::cout << unit.back().id <<  " ";
unit.pop_back();
unit.emplace_back(Unit{2, 3});
std::cout << unit.back().id <<  "";
```

В зависимости от того, смогли ли при реализации вектора задушить агрессивные оптимизации компилятора, такой код может вывести либо `1 2` (всё хорошо), либо `1 1` (компилятор соптимизировал константное поле!).

Компилятор имеет право воспринимать происходящее следующим образом:

- в векторе 1 элемент;
- вектор не реаллоцировался;

- указатель на элемент в первом cout и во втором cout один и тот же;
- и там и там используется константное поле;
- я его уже читал при первом cout;
- зачем мне его читать еще раз, это же константа;
- вывожу закешированное значение.

К сожалению или к счастью, воспроизвести подобное поведение компилятора на практике не получается. Тем не менее вот такой код, который может использоваться для реализации самописных std::optional, по стандарту содержит UB (и не одно!):

```
using storage = std::aligned_storage_t<sizeof(Unit), alignof(Unit)>;
storage s;
new (&s) Unit{1,2};
std::cout << reinterpret_cast<Unit*>(&s)->id << "\n"; // Неопределенное поведение
reinterpret_cast<Unit*>(&s)->~Unit(); // Неопределенное поведение
new (&s) Unit{2,2};
std::cout << reinterpret_cast<Unit*>(&s)->id << "\n"; // Неопределенное поведение
reinterpret_cast<Unit*>(&s)->~Unit(); // Неопределенное поведение
```

Правильный вариант:

```
using storage = std::aligned_storage_t<sizeof(Unit), alignof(Unit)>;
storage s;
auto p = new (&s) Unit{1,2};
std::cout << p->id << "\n";
p->~Unit();
p = new (&s) Unit{2,2};
std::cout << p->id << "\n";
p->~Unit();
```

Но поддерживать указатель, возвращенный оператором new, не всегда возможно. Он занимает место, его надо хранить, что неэффективно при реализации optional: для int32_t будет нужно в три раза больше места на 64-битной системе (4 байта на storage + 8 байт на указатель)!

Поэтому в стандартной библиотеке, начиная с C++17, есть функция "отмывания" невесть откуда взявшихся указателей — std::launder.

```
using storage =
  std::aligned_storage_t<sizeof(Unit), alignof(Unit)>;
storage s;
new (&s) Unit{1,2};
std::cout <<
  std::launder(reinterpret_cast<Unit*>(&s))->id << "\n";
std::launder(reinterpret_cast<Unit*>(&s))->~Unit();
```

```
new (&s) Unit{2,2};
std::cout <<
  std::launder(reinterpret_cast<Unit*>(&s))->id << "\n";
std::launder(reinterpret_cast<Unit*>(&s))->~Unit();
```

Так и при чем тут const? "Настоящая" константность (переменные и поля, объявленные с const) вместе с UB при использовании "неправильных" указателей как раз и позволяют компилятору производить описанные спецэффекты.

Семантика перемещения

Начиная с C++11, у нас есть rvalue-ссылки и семантика перемещения. Причем перемещение недеструктивно: исходный объект остается жив, что порождает множество ошибок. Еще есть проблемы с тем, как избегать накладных расходов при использовании перемещаемых объектов, но с этим можно жить.

Накладные расходы

Несмотря на все громкие заявления, абстракции в C++ имеют далеко не нулевую стоимость. Занятным примером является std::unique_ptr, завязанный на семантику перемещения.

```
void run_task(std::unique_ptr<Task> ptask) {
  // что-то делаем
  ptask->go();
}

void run(...){
  auto ptask = std::make_unique<Task>(...);
  ...
  run_task(std::move(ptask));
}
```

При вызове run_task параметр передается по значению: создается новый объект unique_ptr, а старый остается, но оказывается пустым. Поскольку объектов два, то и вызовов деструктора тоже два. С деструктивной семантикой перемещения[5] вызов деструктора будет только один.

Можно исправить ситуацию — передать по *rvalue*-ссылке:

```
void run_task(std::unique_ptr<Task>&& ptask) {
  // что-то делаем
  ptask->go();
}
```

[5] Например, Rust.

Тогда дополнительного объекта не будет. И произойдет лишь один вызов деструктора. При этом из-за ссылки имеется дополнительный уровень индирекции и обращение к памяти.

Но самое главное: никакого перемещения на самом деле не будет, что может скрыть ошибку в логике программы.

```
void consume_v1(std::unique_ptr<int> p) {}
void consume_v2(std::unique_ptr<int>&& p) {}

void test_v1(){
  auto x = std::make_unique<int>(5);
  consume_v1(std::move(x));
  assert(!x); // ok
}

void test_v2(){
  auto x = std::make_unique<int>(5);
  consume_v2(std::move(x));
  assert(!x); // огонь!
}
```

Запуск второй функции, скорее всего, закончится ошибкой Access Violation (SIGSEGV).

И мы переходим к основной проблеме.

Что случается после

Во-первых, функция `std::move` ничего не делает. Это всего лишь явное преобразование *`lvalue`*-ссылки в *`rvalue`*. Оно никак не влияет на состояние объекта. Обозреваемые эффекты от перемещения могут давать функции, работающие с этой самой *`rvalue`*-ссылкой. В основном это конструкторы и операторы перемещения.

Во-вторых, стандарт C++ не специфицирует состояние, в котором должен остаться объект, из которого произвели перемещение. Оно должно быть валидным в смысле вызова деструктора. Но более ничего не требуется. Объект не обязан быть пустым после перемещения. Его поля не обязаны быть зануленными. Так, у `std::thread` после перемещения нельзя вызывать ни один из методов. А `std::unique_ptr` гарантированно становится пустым (`nullptr`).

Чаще и проще всего натолкнуться на уже перемещенный (moved-out) объект можно при реализации конструкторов, заполняющих поля переданными аргументами, — достаточно дать одинаковые (или почти одинаковые) имена полям и аргументам.

```
struct Person {
public:
  Person(std::string first_name,
         std::string last_name) : first_name_(std::move(first_name)),
                                  last_name_(std::move(last_name)) {
```

```
    std::cerr << first_name; // неверно, use-after-move
  }
private:
  std::string first_name_;
  std::string last_name_;
};
```

Конечно, в таком случае ошибка будет быстро найдена: для `std::string` есть гарантии в наиболее свежих реализациях, что после перемещения объект окажется пустым. Но если сделать конструктор шаблонным и передавать в него тривиально перемещаемые типы, ошибка долго может не проявляться.

```
template <class T1, class T2>
Person(T1 first_name,
       T2 last_name) : first_name_(std::move(first_name)),
                       last_name_(std::move(last_name))
{
  std::cerr << first_name; // неверно, use-after-move
}
...

Person p("John", "Smith"); // T1, T2 = const char*
```

Другой интересный случай использования после перемещения — self-move-assignment, в результате которого из объекта могут внезапно пропадать данные. А могут и не пропадать. В зависимости от того, как реализовали перемещение для конкретного типа.

Например, вот такая наивная реализация алгоритма `remove_if` содержит ошибку:

```
template <class T, class P>
void remove_if(std::vector<T>& v, P&& predicate) {
  size_t new_size = 0;
  for (auto&& x : v) {
    if (!predicate(x)) {
      v[new_size] = std::move(x); // использование после перемещения!
      ++new_size;
    }
  }
  v.resize(new_size);
}
```

Ошибка даст о себе знать в случае, когда элементы контейнера будут содержать поля, не учитывающие возможность самоприсваивания.

```
struct Person {
  std::string name;
  int age;
};
```

```
std::vector<Person> persons = {
  Person { "John", 30 }, Person { "Mary", 25 }
};
remove_if(persons, [](const Person& p) { return p.age < 20; });

for (const auto& p : persons){
  std::cout << p.name << " " << p.age << "\n";
}
```

Скомпилировав и запустив этот код, можно убедиться, что все `name` окажутся пустыми:

```
30
25
```

Отследить некоторые случаи использования после перемещения способны статические анализаторы. Например, в PVS-Studio есть диагностика V1030, а в Clang-Tidy — bugprone-use-after-move.

Если вы реализуете перемещаемые классы и хотите учесть возможность самоприсваивания/самоперемещения, то либо используйте идиому copy/move-and-swap, либо не забывайте проверить совпадение адресов текущего и перемещаемого объектов:

```
MyType& operator=(MyType&& other) noexcept {
  if (this == std::addressof(other)) { // addressof сработает,
                                       // если у вас перегружен &
    return *this;
  }
  ...
}
```

Эллипсис и функции с произвольным числом аргументов

Наверняка все C++ (а уж просто C тем более) программисты знакомы с семейством функций `printf`. Одной из удивительных особенностей этих функций является возможность принимать произвольное число аргументов. А также на `printf` можно писать полноценные программы! Исследованию и описанию этого безумия даже посвящены отдельные статьи[6].

Мы же остановимся только на произвольном числе аргументов. Но для начала я расскажу одну занимательную историю.

[6] Механизм форматных модификаторов функций printf оказался тьюринг-полным. Например, на них можно написать игру крестики-нолики! См. **https://github.com/carlini/printf-tac-toe**.

Некая замечательная библиотека предоставляла красивую функцию:

```
template <class HandlerFunc>
void ProcessBy(HandlerFunc&& fun)
requires std::is_invocable_v<HandlerFunc, T1, T2, T3, T4, T5>;
```

И программист думал вызвать эту восхитительную функцию. В качестве `HandlerFunc` подсунуть лямбду, в которой ему было совершенно наплевать на передаваемые аргументы `T1`, `T2`, `T3`, `T4`, `T5`. Что же он мог сделать?

Вариант первый: честно перечислить пять аргументов с их типами. Как деды делали.

```
ProcessBy([](T1, T2, T3, T4, T5) { do_something(); });
```

Если имена типов короткие, почему бы и нет? Но всё равно как-то слишком подробно. Неудобно. Да и добавится новый аргумент — придется и тут править. Не очень современный C++-подход.

Вариант второй: воспользоваться функциями с произвольным числом аргументов.

```
ProcessBy([](...){ do_something(); });
```

Вау, красота! Компактно и здорово. До чего прогресс дошёл! И оно скомпилировалось. И даже работало. И так программист и оставил.

Но однажды замечательная библиотека обновилась, стала лучше и безопаснее. И начались странные, необъяснимые падения. `SIGILL`, `SIGABRT`, `SIGSEGV`. Все наши любимые друзья хлынули в проект.

Что произошло? Кто виноват? Что делать? Без опытного сыщика тут не обойтись...

Давайте разбираться.

В C можно определять собственные функции, принимающие сколь угодно много аргументов. И сделать это можно двумя способами:

1. Пустой список аргументов.

   ```
   void foo() {
     printf("foo");
   }

   foo(1,2,4,5,6);
   ```

   Казалось бы, функция `foo` не должна в принципе принимать аргументы. Но нет. В C функции, объявленные с пустым списком аргументов, на самом деле являются функциями с произвольным числом аргументов. Действительно ничего не принимающая функция объявляется так:

   ```
   void foo(void);
   ```

   В C++ это безобразие исправили.

2. Эллипсис и `va_list`.

   ```
   #include <stdarg.h>

   void sum(int count, /* Чтобы получить доступ к списку аргументов,
                           нужен хотя бы один явный.*/
            ...) {
   ```

```
    int result = 0;
    va_list args;
    va_start(args, count);
    for (int i = 0; i < count; ++i) // Причем функция не знает,
                                    // сколько аргументов передали.
    {
      result += va_arg(args, int);  // Запрашиваем очередной аргумент.
                                    // Функция не знает, какой у него тип.
                                    // Указываем самостоятельно - int.
    }
    va_end(args);
    return result;
  }
```

Если явного аргумента не будет, то получить доступ к списку остальных нельзя[7]. Более того, мы уйдем в область implementation-defined-поведения.

Также на этот явный аргумент, предшествующий вариативной части, налагаются ограничения:

- он не может быть помечен спецификатором `register`. Но это мало кому надо;
- он не может иметь "повышаемый" тип. Привет нашим любимым `integer/float` promotion. Использовать `float`, `short`, `char` нельзя.

Нарушаем ограничения явного аргумента — получаем неопределенное поведение. Запрашиваем у `va_arg` повышаемый тип — снова неопределенное поведение. Передаём не тот тип, что запрашиваем... правильно, неопределенное поведение.

Невероятные возможности по отстрелу рук и ног себе и пользователям кода! Собственно, на этих возможностях и идёт игра при атаках на `printf`.

И в C++, конечно же, эта прелесть осталась. И не просто осталась, но и значительно усилилась!

C — простой, маленький язык. В нем не так много типов: примитивы, указатели да пользовательские структуры.

В C++ есть ссылки. Есть объекты с интересными конструкторами и деструкторами. И вы уже наверняка догадались, что будет неопределенное поведение, если засунуть ссылку или такой объект в качестве аргумента вариативной функции. Еще больше возможностей для веселой отладки!

Но C++ не был бы самим собой, если бы в нем эту проблему не "решили". Итак, у нас есть вариативные функции в стиле C++:

```
template <class... ArgT>
int avg(ArgT... arg) {
  // Доступно число аргументов.
```

[7] В C23 уже можно. Прогресс! А вы думали, старый-добрый C не развивается?!

```
  const size_t args_cnt = sizeof...(ArgT);
  // Доступны их типы.

  // Итерироваться по аргументам нельзя.
  // Нужно писать рекурсивные вызовы для обработки
  // либо использовать выражения свертки
  return (arg + ... + 0) / ((args_cnt == 0) ? 1 : args_cnt);
}
```

Не очень удобно, но намного лучше и безопаснее.

Ну что ж, теперь, когда все карты открыты, вернемся к нашему детективу.

Убийца — C-вариадик!

```
ProcessBy([](...){ do_something(); });
```

Когда библиотека обновилась, в ней, незначительно на первый взгляд, поменялся один из типов `T`, которые передавались функцией `ProcessBy` в `HandlerFunc`. Но это изменение привело к неопределенному поведению.

А программисту же нужно было использовать C++ вариадик.

```
ProcessBy([](auto...){ do_something(); });
```

Всё. Всего одно слово `auto`, и никто бы не погиб[8]. Удобно.

И, конечно, чтобы не было лишних копирований, надо дописать два амперсанда:

```
ProcessBy([](auto&&...){ do_something(); });
```

Вот теперь всё. Прекрасный способ принять и проигнорировать сколь угодно много аргументов. Ну, а тем программистом был когда-то я сам.

Оператор "запятая"

Если вы начинали свое знакомство с программированием с языков Pascal или C#, то, наверное, знаете, что в них обращение к элементам двумерного массива (а также массивов большей размерности) осуществляется перечислением индексов через запятую внутри квадратных скобок:

```
double [,] array = new double[10, 10];
double x = array[1,1];
```

Также в записи на псевдокоде или в специализированных языках для математических вычислений (MatLab, MathCAD) часто используют именно такой или похожий (круглые скобки) способ.

В C и C++ же на каждую размерность должны быть свои квадратные скобки:

```
double array[10][10];
double x = array[1][1];
```

[8] Можно было бы добавить еще три точки, и все-таки добить ревьюера. Ведь `[](auto...){}` — совмещает C variadic и C++ variadic. И совершенно законная конструкция в C++ до C++26. В грядущей версии нас все-таки решили лишить этой красоты!

Однако написать "неправильно" нам никто не запрещает, и, более того, компилятор обязан это скомпилировать!

```
int array[5][5] = {};
std::cout << array[1, 4]; // Упс!
```

В комбинации с неявным приведением типов и выходами за границы массивов можно наиграть множество неприятностей при невнимательном переносе кода.

Почему это вообще компилируется?

Все дело в операторе "запятая" (`,`). Она последовательно вычисляет оба своих аргумента и возвращает второй (правый).

```
int array[2][5] = {}
auto x = array[1, 4]; // Упс! Это array[4].
// Но для первой размерности максимальное значение = 1.
// Неопределенное поведение!
```

В C++20, на наше счастье, использование оператора "запятая" (`,`) при индексировании массивов пометили как устаревший (deprecated), и теперь компиляторы сыпят предупреждениями (вы всегда можете их превратить в ошибки).

Кстати, облажаться с запятой можно не только при работе с массивами. Например, опечатку[9] можно допустить при написании констант:

```
double A = 1,23; // Упс, A равно 23, а не 1.23.
```

Есть и другие вариации опечаток с запятой. На этом можно было бы и закончить, если бы не один нюанс.

Перегрузки оператора "запятая"

Запятую можно перегрузить и посеять еще больше хаоса.

```
return f1(), f2(), f3();
```

Если "`,`" не перегружена, стандарт гарантирует, что функции будут вызваны последовательно. Если же тут вызывается перегруженная запятая, то до C++17 такой гарантии нет.

В случае встроенной запятой гарантируется, что тип результата совпадает с последним аргументом в цепочке. Если же оператор перегружен, тип может быть каким угодно.

```
auto test() {
  return f1(), f2(), f3();
}

int main() {
  test();
```

[9] На первый взгляд, такая возможность кажется надуманной, но программисты часто копируют что-то откуда-то в свой код. И к сожалению, в мире существуют разные языковые локали: в американской используется точка, а например, во французской — запятая.

```
  static_assert(!std::is_same_v<decltype(f3()), int>);
  static_assert(std::is_same_v<decltype(test()), int>); // ??!
  return 0;
}
```

Запятой часто пользуются в различных шаблонах, чтобы раскрывать пачки аргументов произвольной длины или проверять несколько условий, триггерящих SFINAE.

Из-за потенциальной возможности влететь в перегруженную запятую в выражениях с ней авторы библиотек прибегают к касту каждого аргумента к *void*. Перегрузку, принимающую void, невозможно написать.

```
template <class... F>
void invoke_all(F&&... f) {
  (static_cast<void>(f()), ...);
}

int main() {
  invoke_all([]{
    std::cout << "hello!\n";
  },
  []{
    std::cout << "World!\n";
  });
  return 0;
}
```

Зачем вообще может понадобиться перегружать запятую?

Может быть, для какого-нибудь DSL (domain-specific language).

Или вдруг вам все-таки захочется сделать так, чтоб индексация через запятую работала.

```
struct Index { size_t idx; };

template <size_t N>
struct MultiIndex : std::array<Index, N> {};

template <size_t N, size_t M>
auto operator , (MultiIndex<N> i1, MultiIndex<M> i2) { ... }

template <size_t M>
auto operator , (Index i1, MultiIndex<M> i2) { ... }

template <size_t N>
auto operator , (MultiIndex<N> i1, Index i2) { ... }
```

```
auto operator , (Index i1, Index i2) { ... }

Index operator "" _i (unsigned long long x) {
  return Index { static_cast<size_t>(x) };
}

template <class T, size_t N, size_t M>
struct Array2D {
  T arr[N][M];

  T& operator[] (MultiIndex<2> idx) {
    return arr[idx[0].idx][idx[1].idx];
  }
};

int main() {
  Array2D<int, 5, 6> arr;

  arr[1_i, 2_i] = 5;
  std::cout << arr[1_i, 2_i]; // Ok
  std::cout << arr[1_i, 2_i, 3_i]; // Ошибка компиляции
}
```

Многомерный operator[]

C++23 подарил нам долгожданную возможность перегружать `operator[]` с более чем одним параметром. Любители матриц и Numpy ликуют. C++ очень похорошел за последние годы!

Вместе с долгожданной фичей, разумеется, поставляются новые грабли, которые любезно разложены под разработчиков крупных проектов со смесью стандартов разных версий, от которых бизнес требует релизить фичи, как можно быстрее, так что предупреждения компилятора они могут иногда и проигнорировать...

У вас была библиотека с классом матриц, поддерживающих обращение к отдельным строкам

```
#if __has_include(<span>) && __cplusplus >= 202002L
#include <span>

template <class T>
using Span = std::span<T>;

#else
```

```
template <class T>
struct Span {
    T* data;
    size_t size;

    auto begin() { return data; }
    auto end() { return data + size; }

    auto subspan(size_t ofs, size_t len) {
        return Span<T> {
            data + ofs,
            len
        };
    }
};

#endif

struct Row {
    Span<int> data;

    void operator = (int c) {
        for (auto& x: data) x = c;
    }
};

struct Matrix {
    std::vector<int> data;
    size_t cols;

// Эта перегрузка была у вас 20 лет
    Row operator[](size_t row_idx) {
        return {
            Span<int>{data.data(), data.size()}
                    .subspan(row_idx * cols, cols)
        };
    }

// И вот месяц назад вы добавили восхитительную
// перегрузку для доступа к элементу.
// Библиотека используется с разными версиями C++, так что
// перегрузка под feature-control-флагом работает отлично.
```

```
#ifdef __cpp_multidimensional_subscript
  int& operator[](size_t row_idx, size_t col_idx) {
    return data[row_idx * cols + col_idx];
  }
#endif
};
```

И вот какой-то несчастный из соседней команды использует вашу библиотеку и пишет свой прикладной модуль на C++23, не сильно задумываясь о feature-flags-стандартах. (Вы удивитесь, но это невероятно распространенный сценарий!) И он совершает чудовищную ошибку: помещает определение функции в заголовочный файл

```
auto compute() -> Matrix {
    Matrix m {
        std::vector<int>(12, 0),
        4, // матрица 3 x 4
    };

    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 4; ++j) {
            m[i, j] = i * j;
        }
    }
    return m;
}
```

У него всё работает, всё отлично.

К нему приходит коллега из другой команды и берет его пакет себе, в проект с C++17, и вызывает функцию `compute()`.

У него всё компилируется без каких-либо предупреждений. Запускается. Иногда даже работает. А иногда валится с ошибкой segmentation fault.

Оба программиста идут смотреть вывод санитайзеров и valgrind, изучать core dump, подключаться отладчиком и развлекаться прочими интересными способами отлова багов, чтобы обнаружить:

```
=================================================================
==1==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x604000000040 at pc
0x000000401923 bp 0x7ffe36f379e0 sp 0x7ffe36f379d8
WRITE of size 4 at 0x604000000040 thread T0
    #0 0x401922 in Row::operator=(int) /app/example.cpp:38
    #1 0x401922 in compute() /app/example.cpp:68
    #2 0x40116b in main /app/example.cpp:75
    #3 0x749978829d8f  (/lib/x86_64-linux-gnu/libc.so.6+0x29d8f) (BuildId:
962015aa9d133c6cbcfb31ec300596d7f44d3348)
```

```
    #4 0x749978829e3f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x29e3f)
(BuildId: 962015aa9d133c6cbcfb31ec300596d7f44d3348)
    #5 0x401274 in _start (/app/output.s+0x401274) (BuildId:
18a2ccdd2845a7b183f1249366db2569b4a1d038)
```

Ну да, ведь до C++23 была только одна перегрузка `operator[]`.

В строке `m[i, j] = i * j;` вместо многомерного оператора вызывается одномерный, в который передается результат `operator ,` , т. е. второй элемент. Вот и приплыли с выходом за границы буфера, если у матрицы столбцов больше чем строк. Не говоря уже о том, что код в принципе теперь сломан и делает что-то другое.

Ах да! Обещанное предупреждение компилятора... Есть предупреждение, да. Но по умолчанию только в C++20.

```
<source>: In function 'Matrix compute()':
<source>:68:16: warning: top-level comma expression in array subscript is deprecated
[-Wcomma-subscript]
   68 |             m[i, j] = i * j;
```

В C++17 и ранее — нет. А с ключом `-Wall` есть, но другое:

```
<source>:68:15: warning: left operand of comma operator has no effect [-Wunused-value]
   68 |             m[i, j] = i * j;
```

Но `-Wunused-value` бывает слишком "шумным", особенно в легаси-проектах, и его часто отключают.

Веселого вам перехода с C++17 на C++23, минуя C++20!

try-блок функций

В C++ существует альтернативный синтаксис для определения тела функции, позволяющий навесить на него целиком перехват и обработку исключений.

```
// Стандартный способ
void f() {
  try {
    may_throw();
  } catch (...) {
    handle_error();
  }
}

// Альтернативный синтаксис
void f() try {
  may_throw();
} catch (...) {
  handle_error();
}
```

Во-первых, запись становится короче, с меньшим уровнем вложенности. Во-вторых, эта фича позволяет нам ловить исключения там, где стандартным способом это сделать невозможно: в списке инициализации класса, при инициализации подобъекта базового класса и подобном.

```
struct ThrowInCtor {
  ThrowInCtor() {
    throw std::runtime_error("err1");
  }
};

struct TryStruct1 {
  TryStruct1() try {

  } catch (const std::exception& e) {
    // Будет поймано исключение из конструктора `c`.
    std::cout << e.what() << "\n";
  }
  ThrowInCtor c;
};

struct TryStruct2 {
  TryStruct2() {
    try {

    } catch (const std::exception& e) {
      // Исключение не будет поймано,
      // поскольку тело конструктора
      // исполняется после инициализации полей.
      std::cout << e.what() << "\n";
    }
  }
  ThrowInCtor c;
};
```

На примере с try-блоком для конструктора мы сталкиваемся, на первый взгляд, со странной неожиданностью: несмотря на блок catch, исключение вылетает в код, вызывающий конструктор, — и код выше печатает:

```
err1
something wrong
something wrong
```

Это логично, ведь если при инициализации полей класса вылетело исключение, мы никак не можем исправить ситуацию и починить объект.

Потому можно иногда встретить такие страшные нагромождения:

```
struct S {
  S(...) try :
      a(...),
      b(...) {
    try {
      init();
    } catch (const std::exception& e) {
      log(e);
      try_repair();
    }
  } catch (const std::exeption& e) {
    // Не получилось починить или
    // неисправимая ошибка в полях.
    log(e);
    // Неявный повторный выброс ошибки
  }

  A a;
  B b;
};
```

Ну хорошо. А как насчет деструкторов? Ведь из деструкторов крайне нежелательно выкидывать исключения, и возможность красиво и просто поставить `catch`, который бы гарантированно перехватил всё, весьма недурна.

```
struct DctorThrowTry {
  ~DctorThrowTry() try {
    throw std::runtime_error("err");
  } catch (const std::exception& e) {
    std::cout << e.what() << "\n";
  }
};
```

Выглядит неплохо. Но у нас C++, так что это не работает!

Кто-то очень доброжелательный решил, что в случае с деструкторами поведение по умолчанию должно быть таким же, как и с конструкторами. То есть **`catch`-блок деструктора неявно прокидывает исключение дальше**. И привет всем возможным проблемам с исключениями из деструкторов, в том числе нарушению неявного `noexcept(true)`.

Однако, в отличие от конструкторов, для деструкторов добавили возможность подавить неявное пробрасывание пойманного исключения.

Для этого нужно всего лишь... добавить return!

```
struct DctorThrowTry {
  ~DctorThrowTry() try {
     throw std::runtime_error("err");
  } catch (const std::exception& e) {
    std::cout << e.what() << "\n";
    return; // Исключение не будет перевыброшено!
  }
};
```

Удивительно, но из-за этого в C++ есть случай, в котором return последней командой в void-функции меняет ее поведение.

Также нужно добавить, что в catch-блоке деструкторов и конструкторов нельзя обращаться к нестатическим полям и методам класса — будет неопределенное поведение. По понятным причинам. В момент входа в catch-блок они все уже мертвы.

```
struct S {
  A a;
  B b;

  S() try {
    ...
  } catch (...) {
     do_something(a); // Неопределенное поведение!
  }

  ~S() try {
    ...
  } catch (...) {
    do_something(b); // Неопределенное поведение!
    return;
  }
};

// Но при этом

bool fun(T1 a, T2 b) try {
  ...
  return true;
} catch (...) {
  // Важно: этот блок не ловит исключения,
  // возникающие при инициализации a и b.
```

```
  do_something(a); // Ok!
  return false;
}
```

Подведем итог.

- Для обычных функций и `main` с помощью альтернативного синтаксиса можно удобно и красиво перехватывать все исключения, которые могли бы вылететь. И поведение по умолчанию — именно перехват. Дальше не летит.
- Для конструкторов можно ловить исключения из конструкторов полей, обрабатывать их (печатать в лог), но подавить нельзя. Либо кидаете своё, новое исключение, либо пойманное неявно будет проброшено дальше.
- Для деструкторов также будет неявный проброс, но его можно подавить, добавив `return`.

Типы нулевого размера

В C++ при определении собственных классов и структур никто нам не запрещает не указывать ни одного поля, оставляя структуру пустой:

```
struct MyTag {};
```

Конечно же, мы можем не только объявлять пустые структуры, но и создавать объекты этих типов.

```
struct Tag {};

Tag func(Tag t1) {
  Tag t2;
  return Tag{};
}
```

Возможности, несомненно, полезные и широко используемые:

- для определения абстрактного статического или динамического полиморфного интерфейса;
- для введения тегов выбора нужной перегрузки;
- для определения различных предикатов и метафункций над типами.

А давайте сыграем в игру? Я буду показывать вам разные определения структур, а вы постараетесь угадать их размеры в байтах (`sizeof`). Начинаем?

```
struct StdAllocator {};

struct Vector1 {
  int* data;
  int* size_end;
```

```
  int* capacity_end;
  StdAllocator alloc;
};
struct Vector2 {
  StdAllocator alloc;
  int* data;
  int* size_end;
  int* capacity_end;
};


struct Vector3 : StdAllocator {
  int* data;
  int* size_end;
  int* capacity_end;
};
```

Угадали?

`Vector1` и `Vector2` имеют размеры `4*sizeof(int*)`. Но как же так?! Откуда берутся `3*sizeof(int*)`, совершенно очевидно. Но четвертый-то откуда?!

Все очень просто: в C++ не бывает структур нулевого размера. И потому размер пустой структуры `sizeof(StdAllocator) == 1`.

Но `sizeof(int*) != 1`. По крайней мере на *x*86. А это еще проще: выравнивание и паддинг. `Vector1` дополняется байтами в конце, чтобы его размер был кратен выравниванию первого поля. А в `Vector2` дополняется байтами между `alloc` и `data`, чтобы смещение до `data` было кратным его выравниванию. Всё очень просто и очевидно! Если же вам, как и многим другим людям, которые не задаются подобными вопросами каждый день, неочевидно наличие паддинга в той или иной структуре, то советую использовать флаг `-Wpadded` для компиляторов GCC/Clang.

Хорошо, мы разобрались с `Vector1` и `Vector2`. А что там с `Vector3`? Тоже `4*sizeof(int*)`? Ведь мы же знаем, что подобъект базового класса должен быть где-то размещен, а его размер, как мы выяснили, ненулевой... А вот и нет! Размер `Vector3` равен `3*sizeof(int*)`! Но как же так?! А это называется EBO (empty base optimization) — оптимизация пустого базового класса.

Интересный zero-cost! Для сравнения можно глянуть на аналогичные пустые структуры в Rust. Там их размер может быть равен нулю.

Ну ладно, мы выяснили, что, неаккуратно использовав пустые структуры, мы можем получить увеличение потребления памяти. Давайте играть дальше.

```
struct StdAllocator {};
struct StdComparator {};
```

```
struct Map1 {
  StdAllocator alloc;
  StdComparator comp;
};

struct Map2 {
  StdAllocator alloc;
  [[no_unique_address]] StdComparator comp;
};

struct Map3 {
  [[no_unique_address]] StdAllocator alloc;
  [[no_unique_address]] StdComparator comp;
};

struct MapImpl1 : Map1 {
  int x;
};

struct MapImpl2 : Map2 {
  int x;
};

struct MapImpl3 : Map3 {
  int x;
};
```

Чему равны размеры `Map1`, `Map2`, `Map3`?

Ну, тут всё просто:

- очевидно, что `sizeof(Map1) == 2`, ведь она состоит из двух пустых структур, каждая из которых имеет размер 1;
- благодаря атрибуту `[[no_unique_address]]` из стандарта C++20 (Clang поддерживает с C++11) `Map2` и `Map3` должны иметь размер 1. В `Map3` оба поля разделяют общий адрес. В `Map2` то же самое. Да и меньше, чем 1, не бывает.

Хорошо. А что же теперь с наследующими структурами?

Все по `2*sizeof(int)`? А вот и нет: у `MapImpl3` работает EBO!

Ну и ладно. В этом есть какая-то логика и закономерность. Это еще можно принять. Хотя... На самом деле, вы были правы! Ведь если у вас компилятор MSVC, то `[[no_unique_address]]` просто не работает. И не будет работать. Потому что MSVC долгое время просто игнорировал незнакомые ему атрибуты. И если поддержать

[[no_unique_address]], то сломается бинарная совместимость. Используйте [[msvc::no_unique_address]]! EBO, правда, пока не работает.

Массив нулевого размера

Язык C (не C++), начиная с версии стандарта 99, позволяет использовать следующую любопытную конструкцию:

```
struct ImageHeader{
  int h;
  int w;
};

struct Image {
  struct ImageHeader header;
  char data[];
};
```

Поле data в структуре Image имеет нулевой размер. Это FAM (flexible array member, "гибкий" элемент массива). Очень удобная штука, чтобы получать доступ статически к массиву неизвестной длины, размещенному сразу после некоторого заголовка в бинарном буфере. Длина массива обычно указывается в самом заголовке. FAM может быть только последним полем в структуре.

Стандарт C++ такие фичи не разрешает. Но ведь есть GCC с его нестандартными включенными по умолчанию расширениями.

Что будет, если сделать так?

```
struct S {
  char data[];
};
```

Чему будет равен размер структуры S?

В стандартном C пустые структуры в принципе запрещены, и поведение программы с ними не определено. GCC определяет их размер нулевым при компиляции C-программ. А при компиляции C++ размер, как мы выяснили ранее, единичный. Дело пахнет страшными багами и ночными кошмарами при неосторожном проектировании C++-библиотек с "сишным" интерфейсом или использованием C-библиотек в C++!

Но вернемся все-таки к нашей структуре с FAM. Поле в ней есть. Стандартный C опять-таки требует, чтобы было еще хотя бы одно поле ненулевой длины перед FAM. GNU C же охотно сделает нам структуру нулевого размера.

А теперь посмотрим на GCC C++:

```
struct S1 {
  char data[];
};
```

```
struct S2 {};

static_assert(sizeof(S1) != sizeof(S2));
static_assert(sizeof(S1) == 0);
```

И вот уже внезапно у нас в C++ структуры нулевого размера. Только C++ нестандартный. О том, каким образом такие структуры будут взаимодействовать с EBO, нужно читать в спецификации к GCC.

Теговая диспетчеризация[10]

Мы видели, что неаккуратное использование пустых структур приводит к увеличению размера других, непустых, структур. А может, еще есть какие-то подводные камни? Например, при использовании пустых структур-тегов для выбора перегрузки?

Есть ли разница между:

```
struct Mul {};
struct Add {};

int op(Mul, int x, int y) {
  return x * y;
}

int op(Add, int x, int y) {
  return x + y;
}
```

и

```
int mul(int x, int y) {
  return x * y;
}

int add(int x, int y) {
  return x + y;
}
```

в плане генерируемого кода?

Краткий ответ: "Да, есть разница". Но зависит от конкретной имплементации. Стандарт не гарантирует оптимизацию пустых аргументов. От перемены позиций тегов может меняться бинарный интерфейс. Поиграться с наиболее заметными изменениями можно на примере MSVC.

[10] Tag dispatching.

Перегрузки с zero-size-тегами	Две разные функции без тегов

x$ = 8
y$ = 16
__formal$ = 24
int op(int,int,Mul) PROC
  imul ecx, edx
  mov eax, ecx
  ret 0
int op(int,int,Mul) ENDP
 	
x$ = 8
y$ = 16
__formal$ = 24
int op(int,int,Add) PROC
  lea eax, DWORD PTR [rcx+rdx]
  ret 0
 	
int op(int,int,Add) ENDP
	_x$ = 8
	_y$ = 12
	
	int mul(int,int) PROC
	  mov eax, DWORD PTR _x$[esp-4]
	  imul eax, DWORD PTR _y$[esp-4]
	  ret 0
	int mul(int,int) ENDP
	
	_x$ = 8
	_y$ = 12
	
	int add(int,int) PROC
	  mov eax, DWORD PTR _x$[esp-4]
	  add eax, DWORD PTR _y$[esp-4]
	  ret 0
	int add(int,int) ENDP

Послесловие к оптимизации размеров структур

Простой перестановкой полей можно уменьшать размер структур. Например, на классической 32-битной архитектуре размер этой структуры из-за выравнивания равен 16 байтам:

```
struct A {
  int  x;
  char foo_x;
  int  y;
  char foo_y;
};
```

А вот этой — уже 12 байт:

```
struct A {
  int  x;
  int  y;
  char foo_x;
  char foo_y;
};
```

Подобные оптимизации не нужны, если объекты создаются штучно, и не всегда возможны, если структуры отображают некую внешнюю структуру данных.

Однако если объекты создаются миллионами, то можно заметно оптимизировать потребление памяти простейшим рефакторингом. Единственная проблема — не всегда сразу видно, какие структуры можно оптимизировать, а какие — нет. Тем более, что на различных архитектурах используются разные размеры данных и действуют разные правила выравнивания. Жизнь облегчают анализаторы кода. Например, в PVS-Studio для этой задачи есть диагностика V802.

Если от структуры никак не избавиться, рекомендую обратить внимание на всё более популярную технику: преобразование массива структур (AoS, Array-of-Structs) в структуру из массивов (SoA, Struct-of-Arrays).

(Не)явное приведение типов

Рассматривая графики метрик, отображающие число ожидающих обработки запросов в момент времени, дежурный инженер заметил, что с графиком что-то не так: отсутствие какого-либо пульса, просто горизонтальная прямая. Взглянув на ось *y*, инженер увидел, что график замер на значении 18446744073709552000. Такому вселенскому масштабу трафика могли бы позавидовать все крупные компании. Но, разумеется, никакого трафика не было. А была ошибка в метрике, даже целых две. Ну и чтобы все любители C++ порадовались, скажу, что этот код был написан на Rust.

Первая ошибка родилась от небрежно написанного кода подсчета запросов в очереди. Вместо того, чтобы возложить эту обязанность на пару "конструктор/деструктор" и гарантировать ровно один инкремент в начале и один декремент в конце, программист решил пойти старыми дедовскими путями и выполнять декременты в разных ветках разных вложенных условных операторов. В результате иногда вычитание происходило дважды, приводя к переполнению беззнакового счетчика. В процессе детального разбора выяснились и другие, более серьезные проблемы, но они не имеют никакого отношения к теме главы.

Произошло переполнение. Хорошо. Счетчик, похоже, 64-битный, раз такое большое число. Но постойте. Беззнаковый –1 в `uint64` — это 18446744073709551615. А число, которое увидел инженер, немного больше...

Код, генерирующий метрики, был следующего вида:

```
metrics.set(Metric::InflightRequests, counter as _);
```

Очевидно, в деле было замешано приведение типов. Второй аргумент метода `set` ожидал тип `f64`.

Любознательный читатель должен поглядеть в стандарт IEEE 754 и найти разгадку магии чисел. Для менее любознательного читателя скажу лишь, что `f64(u64(-1)) == f64(u64(-1024))`.

```
counter as _
```

Явное преобразование к чему-то непонятному, известному из контекста, но совсем неочевидному при чтении. Полезная и сомнительная фича Rust.

Теперь мы можем вернуться к C++. В C++ приведения тривиальных типов не только происходят неявно, но еще и иногда приводят к неопределенному поведению. Поэтому к вопросу нужно подойти максимально серьезно.

Итак, вы — автор библиотеки. Вы хотите сделать ее максимально надежной, как можно более защищенной от дурака и как можно более дружелюбной к пользователю, чтобы компилятор смог направить его к единственно правильному варианту использования.

Приняв во внимание фиаско с кодом на Rust, которое я вам только что описал, вы сразу же решаете прибегнуть к помощи strong typedefs (впрочем, в случае Rust-кода они бы тоже помогли).

```
// Тут вы написали очень длинный и развернутый комментарий о том,
// что значения имеют тип double (f64), потому что так надо.
// И пользователь должен иметь в виду сопутствующие ограничения.
// И все такое прочее...
struct MetricSample{
  // Чтобы избежать неявного приведения, вы сразу же
  // добавили explicit, как советуют все best practices.
  explicit MetricSample(double val): value {val} {}
private:
  double value;
};

class Metrics {
public:
  // Отлично, теперь у пользователя нет иного варианта, как выполнить
  // явное преобразование к MetricSample,
  // а там уж он почитает документацию...
  void set(std::string_view metric_name, MetricSample val);
};

// Вы пишете UX-тест.
int main() {
  uint64_t value = -1;
  Metrics m;
  m.set("InflightRequests", value);
  m.set("InflightRequests" MetricSample{value});
}
```

И он не компилируется, как вы и хотели.

Ошибки компиляции

```
<source>:23:31: error:
no viable conversion from 'uint64_t' (aka 'unsigned long') to
'MetricSample'
   23 |     m.set("InflightRequests", value);
      |                               ^~~~
<source>:4:8: note: candidate constructor (the implicit copy constructor)
not viable: no known conversion from 'uint64_t' (aka 'unsigned long')
to 'const MetricSample &' for 1st argument
    4 | struct MetricSample{
      |        ^~~~~~~~~~~
<source>:4:8: note: candidate constructor (the implicit move constructor)
not viable: no known conversion from 'uint64_t' (aka 'unsigned long')
to 'MetricSample &&' for 1st argument
    4 | struct MetricSample{
      |        ^~~~~~~~~~~
<source>:7:14: note: explicit constructor is not a candidate
    7 |     explicit MetricSample(double val): value {val} {}
      |              ^
<source>:16:57: note: passing argument to parameter 'val' here
   16 |     void set(std::string_view metric_name, MetricSample val);
      |                                                         ^
<source>:24:30: error: expected ')'
   24 |     m.set("InflightRequests" MetricSample{value});
      |                              ^
<source>:24:10: note: to match this '('
   24 |     m.set("InflightRequests" MetricSample{value});
      |          ^
```

Отлично. Дело сделано. Релизим.

Через неделю к вам приходит опытный пользователь и говорит, что отстрелил себе ногу вашей библиотекой.

Ваша защита от неявного приведения типов не содержит защиты от опытного дурака:

```
int main() {
  uint64_t value = -1;
  Metrics m;
  m.set("InflightRequests", MetricSample(value));
}
```

И код компилируется.

И тут вы, давно наслаждающиеся всеми прелестями современнейшего C++, совершенно безопасного в прямых руках с C++ Core Guidlines, вспоминаете про эту проклятую разницу между круглыми и фигурными скобками при вызове конструкторов, хватаетесь за голову и начинаете думать, как спасти вашего пользователя от него самого.

Решение есть! Спасибо, C++20!

```
#include <concepts>

struct MetricSample{
  // Теперь только double может быть передан.
  // Никаких неявных преобразований, поскольку это шаблон.
  explicit MetricSample(std::same_as<double> auto val) :
    value {val} {}
private:
  double value;
};

int main() {
  uint64_t value = -1;
  Metrics m;
  m.set("InflightRequests", MetricSample(value));
  m.set("InflightRequests", MetricSample{value});
  m.set("InflightRequests", value);
}
```

Теперь ничего не компилируется.

Всё? Нет, подождите. Это же C++, а не у всех есть C++20! Вот версия для C++14 и C++17. Даже для C++11 можно сделать (можете взять это в качестве домашнего задания).:

```
#include <type_traits>

struct MetricSample{
  // Теперь только double может быть передан.
  // Никаких неявных преобразований, поскольку это шаблон.
  template <typename Double,
            typename = std::enable_if_t<std::is_same_v<Double, double>>
            >
  explicit MetricSample(Double val): value {val} {}
private:
  double value;
};
```

Время идет. Ваша библиотека набирает популярность. В какой-то момент к вам приходит пользователь и говорит: "Хотелось бы мне еще добавлять комментарий к значению метрики".

Не вопрос! Вы решаете добавить перегрузку метода set с третьим — строковым — параметром.

```
class Metrics {
public:
  // Чтобы сделать явной для пользователя необходимость
  // аллоцировать память под строку и не делать лишних неявных
  // копий, вы решаете использовать rvalue reference.
  // Ведь это отличный способ продемонстрировать, что ваш
  // интерфейс желает заполучить владение строкой.
  // И пользователь будет должен выполнить явный move.
  void set(std::string_view metric_name, MetricSample val,
           std::string&& comment);
}

int main() {
  Metrics m;
  auto val = MetricSample(1.0);
  std::string comment = "comment";
  m.set("MName", val, comment); // не компилируется, как и хотели
  m.set("MName", val, "comment"); // сомнительно, но для удобства Ok
  m.set("MName", val, std::move(comment));
  m.set("MName", val,
        std::string_view("comment")); // не компилируется, хорошо
  auto gen_comment = []()->std::string { return "comment"; };
  m.set("MName", val, gen_comment()); // отлично
}
```

Всё хорошо. Релизим. Через два дня к вам приходит пользователь и говорит, что он отстрелил себе ногу вашей библиотекой. И показывает **это**:

```
int main() {
  Metrics m;
  auto val = MetricSample(1.0);
  m.set("Metric", val, 0);
}

Output:
terminate called after throwing an instance
of 'std::logic_error' what():
basic_string: construction from null is not valid
Program terminated with signal: SIGSEGV
```

В этот момент вы проклинаете класс std::string, неявную интерпретацию 0 как указателя, а также пользователя, который совершенно не читает не только документацию, но и вообще код, который написал. Вы справляетесь с желанием написать собственный класс строк и начинаете думать, как подстелить соломку и в этом случае.

Здесь, конечно, начинаются самые разные варианты. Мы можем разрешить только rvalue string:

```
class Metrics {
public:
  // Только rvalue-ссылки на string. Никакого неявного приведения типов
  void set(std::string_view metric_name,
           MetricSample val,
           std::same_as<std::string> auto&& comment) {};
};

int main() {
  Metrics m;
  auto val = MetricSample(1.0);
  std::string comm = "comment";
  m.set("Metric", val, comm); // не компилируется
  m.set("Metric", val, 0);    // не компилируется
  m.set("Metric", val, std::move(comm)); // компилируется, как и хотели
  m.set("MName", val,
        std::string_view("comment"));    // не компилируется, хорошо
  auto gen_comment = []()->std::string { return "comment"; };
  m.set("MName", val, gen_comment()); // отлично
}
```

Но вы уже живете в проклятом мире: вы разрешали использовать строковые литералы напрямую. Если их запретить, пользователь расстроится — у него код перестанет компилироваться. Так что придется добавить и их. А чтобы пользователь не совал нулевые указатели и использовал только строковые литералы, есть замечательное решение — ссылки на массивы! Ведь строковые литералы — это массивы...

```
class Metrics {
public:
  // Только строковые литералы и явная передача владения
  // строкой разрешены вашим интерфейсом.
  void set(std::string_view metric_name, MetricSample val,
           std::same_as<std::string> auto&& comment) {};
  template <size_t N>
  void set(std::string_view metric_name, MetricSample val,
           const char(&comment)[N]) requires (N > 0) {
```

```
    this->set(metric_name, val, std::string(comment, N-1));
  }
};

int main() {
  Metrics m;
  auto val = MetricSample(1.0);
  std::string comm = "comment";
  const char* null_comment = 0;
  m.set("Metric", val, "comment"); // "ok"
  m.set("Metric", val, null_comment); // не компилируется
  m.set("Metric", val, comm); // не компилируется
  m.set("Metric", val, 0); // не компилируется
  m.set("Metric", val, std::move(comm)); // компилируется, как и хотели
  m.set("MName", val,
        std::string_view("comment")); // не компилируется, хорошо
  auto gen_comment = []()->std::string { return "comment"; };
  m.set("MName", val, gen_comment()); //  работает отлично
}
```

Всё отлично. Релизим!

В какой-то момент особенно ушлый пользователь сконструирует кривой массив и передаст его вместо строкового литерала... Но в этот момент единственное, что вы сможете сделать, — это проигнорировать такого пользователя. Поскольку в безрезультатных попытках выразить и такое ограничение в C++ (и оставить прием литералов без изменений пользовательского кода), можно сойти с ума.

В заключение стоит добавить, что в C++23 появилось новое применение для ключевого слова `auto`:

```
void call_it(auto&& obj) {
  call_impl(auto(obj));
}
```

Я видел разработчиков, которым приходится много работать одновременно с Rust и C++, и эта новая фича для них выглядит как преобразование `obj` к какому-то типу, который указан как аргумент `call_impl`. Прямо как `as _` или вызов `Into::into()` в Rust. Это могло бы быть очень логичным...

Но нет, это совершенно другая фича. Компиляторы C++ не делают таких сложных выводов типов: `auto` в этой позиции нужен, чтобы создавать копии, не имея под рукой имени типа.

Неявное приведение к bool

Вы пишете новую восхитительную библиотеку сериализации в JSON. Для этого у вас уже написано много своих версий функции `stringify` для поддерживаемых JSON типов. Их там немного...

И вот у вас есть

```
auto stringify(bool b) -> std::string_view {
  return b ? "true" : "false";
}

auto stringify(std::string_view s) -> std::string_view {
  return s;
}
```

Выглядит хорошо и логично.

Вы тестируете эти функции

```
int main() {
  std::cout << stringify(true) << "\n";
  std::cout << stringify("string") << "\n";
}
```

и получаете

```
true
true
```

Удивлены? Но тут нет ничего удивительного! Просто строковый литерал, который имеет тип `const char[7]`, неявно приводится к `const char*`, который неявно приводится к `bool`. А поскольку это всё built-in-преобразования, они имеют приоритет перед user-defined-преобразованием к `std::string_view` через его конструктор.

С неявным приведением указателей к `bool` есть еще известный дефект в инициализаторе через фигурные скобки:

```
bool array[5] = {true, false, true, false, true};
std::vector<bool> vector {array, array + 5};
std::cout << vector.size() << "\n";
```

Будет выведено `2`, а не `5`, потому что указатели неявно приводятся к `bool`! Дефект кое-как исправили в C++20. Теперь Clang отказывается это компилировать, а GCC просто выдает предупреждение.

```
<source>:9:32: error: type 'bool[5]' cannot be narrowed to 'bool' in initializer list
[-Wc++11-narrowing]
    9 |     std::vector<bool> vector { array, array + 5};
      |                                ^~~~
<source>:9:32: note: insert an explicit cast to silence this issue
    9 |     std::vector<bool> vector { array, array + 5};
      |
```

А вы знаете, как определить для вашего типа все возможные арифметические операторы и операторы сравнения разом? Нужно всего лишь определить неявный оператор приведения к bool, конечно же!

```
struct OptionalPositive {
  int x;

  operator bool() const {
    return x >= 0;
  }
};

int main() {
  std::cout << 5 + OptionalPositive { 5 };
  std::cout << (5 < OptionalPositive { 5 });
  std::cout << (5 == OptionalPositive { 5 });
  std::cout << (5 * OptionalPositive { 5 });
}
```

Это компилируется и выдает результат 6005. Потому как выполняется user-defined неявное приведение к bool, который далее неявно приводится к int. Всё правильно.

Последние версии Clang хотя бы выдают частично предупреждения:

```
<source>:13:21: warning: result of comparison of constant 5 with expression of type
'bool' is always false [-Wtautological-constant-out-of-range-compare]
   13 |     std::cout << (5 < OptionalPositive { 5 });
      |                   ~ ^ ~~~~~~~~~~~~~~~~~~~~~
<source>:14:21: warning: result of comparison of constant 5 with expression of type
'bool' is always false [-Wtautological-constant-out-of-range-compare]
   14 |     std::cout << (5 == OptionalPositive { 5 });
      |
```

Правда, если поменять тип константы слева на double, в Clang 19 предупреждение исчезнет. Но компилироваться оно не перестанет.

Никогда, если только у вас не C++98, не определяйте неявный operator bool! Он всегда должен быть explicit. Если вы боитесь, что это заставит вас делать static_cast<bool> там, где этого не хочется делать, то не переживайте! C++ определяет несколько контекстов, в которых explicit operator bool все равно может быть вызван неявно: в условиях if, for и while, а также в логических операциях. Этого достаточно для большинства использований operator bool.

Если у вас C++98... Я вам очень соболезную. Но и даже в вашем печальном случае есть решение. Чудовищно громоздкое, но решение — можете ознакомиться с устаревшей Safe Bool Idiom в свободное время в качестве домашнего задания.

Если коротко, вместо operator bool предлагалось определить

```
// Указатель на метод в приватном классе!
typedef void (SomePrivateClass::*bool_type) () const;
operator bool_type(); // неявное приведение к этому указателю
```

И тогда ваш объект в условных операциях неявно приводился бы к указателю, а указатель далее неявно приводился бы к bool.

Пользовательские операторы сравнения в C++20

C++, конечно, развивается медленным циклом в три года. Чтобы ничего не сломать. Большой успех — большая ответственность... Но несмотря на всю медлительность и осторожность, новые стандарты C++ все равно умудряются подложить мину там, где никто не ожидает.

C++20 добавил долгожданный "оператор НЛО" <=>, three-way-comparison, позволяющий существенно сократить однообразный код для определения операций сравнения над пользовательскими типами.

```
struct Pair {
  int x;
  int y;

  auto operator<=>(const Pair&) const = default;
  // Все операции <, >, ==, !=, <=, >= -- выведены автоматически.
  // Покомпонентное сравнение в порядке объявления полей!
};

// И вот мы уже можем сравнивать точки!
Pair {1, 2} < Pair { 2, 3 };
```

Ну почти.

```
<source>:8:10: error: 'strong_ordering' is not a member of 'std'
    8 |     auto operator <=> (const Pair&) const = default;
      |          ^~~~~~~~
<source>:1:1: note: 'std::strong_ordering' is defined in header '<compare>'; this is
probably fixable by adding '#include <compare>'
```

Внезапно... Вы обязаны подключить заголовок, если хотите использовать новую синтаксическую конструкцию с автоматической реализацией (через =default). Причем узнаете вы об этом, только когда попробуете использовать сравнение.

Добавим заголовок, и всё будет работать. Здорово? Конечно же! Но есть кое-что еще.

Не все типы можно осмысленно упорядочивать. Иногда достаточно только равенства. В C++20 можно определить operator ==, и operator != будет выведен автоматически.

Более того, реализация через = default также работает.

```
struct Pair {
  int x;
  int y;

  bool operator==(const Pair&) const = default;

};

// operator != выведен автоматически
Pair {1, 2} != Pair{3, 4};
```

Другие операторы сравнения тоже можно автоматически определять по умолчанию.

Прекрасно. Но сравнениями для одного и того же типа всё не заканчивается. Иногда нам нужно уметь сравнивать объекты разных типов, например std::string и std::string_view.

Как разработчики справлялись с этой задачей до C++20?

Ну, например, эксплуатируя неявное приведение типов, когда это уместно.

```
struct String;

struct StringView {
  // Разрешаем неявное приведение, ведь это удобно
  StringView(const String&) {}
  bool operator==(const StringView &) const { return true; }
};

struct String {
  bool operator==(const StringView &sv) const {
    // А тут меняем порядок местами, ведь у StringView уже есть operator ==.
    // На этапе выбора перегрузки компилятор найдет его по первому аргументу,
    // а ко второму (*this: String) можно применить неявное приведенине типа.
    // Все отлично!
    return sv == *this;
  }
};

String{} == String{};
```

Работает, компилируется с C++17, всё отлично!

А без трюков 4 перегрузки нужны...

Ну хорошо. Переходим в C++20.

```
Program returned: 139
Program terminated with signal: SIGSEGV
```

Шикарно! Надеюсь, если вы компилировали без `-Werror`, у вас были хотя бы тесты.

```
<source>:14:19: warning: in C++20 this comparison calls the current function
recursively with reversed arguments
   14 |         return sv == *this;
      |                ~~~^~~~~~~~
<source>: In function 'int main()':
<source>:19:31: warning: C++20 says that these are ambiguous, even though the second
is reversed:
   19 |     return String{} == String{};
```

C++20 позаботился о разработчиках. И теперь им не нужно выдумывать странные перестановки аргументов местами, чтобы писать поменьше перегрузок операторов сравнения. C++20 ввел правила переписывания всех операторов сравнения, так что компиляторы выполнят перестановку за вас. Даже если в ней нет надобности. И, разумеется, веселые и находчивые разработчики старых кодовых баз получат бесконечную рекурсию. А с ней и неопределенное поведение. И SIGSEGV от переполнения стека, если повезет.

Эти изменения в правилах поиска перегрузок для операций сравнения имели и другие побочные эффекты. Их постарались исправить в предложении P2468R2 The Equality Operator You Are Looking For. Оно принято и реализовано в C++23.

Но внезапную рекурсию все равно не убрали! Полагайтесь на предупреждения компилятора.

В C++20/23, если у вас есть неявное приведение между типами, не нужно больше ничего выдумывать — определяйте операции для одного и того же типа, а комбинации будут получены автоматическими перестановками.

```
struct String {
  bool operator==(const String&) const = default;
};
struct StringView {
  StringView(const String&) {}
  bool operator==(const StringView &) const = default;
};

String{} == StringView{String{}};
StringView{String{}} == String{};
```

Если неявного приведения нет, то достаточно определить только одну дополнительную перегрузку:

```
struct String {
  bool operator==(const String&) const = default;
};
```

```
struct StringView {
  explicit StringView(const String&) {}
  bool operator==(const StringView &) const = default;
};

bool operator == (const String& s, const StringView& sv) {
  return sv == StringView{s};
}

// Обе перестановки работают
String{} == StringView{String{}};
StringView{String{}} == String{};
```

Атрибут [[assume]]

"Есть некоторая вселенская несправедливость", — подумали в комитете стандартизации C++. — Мы так много всего в языке назначили быть неопределенным поведением, чтобы помочь компиляторам генерировать оптимальный код. Но не дали такую же стандартную возможность нашим пользователям — программистам!"

Да, C++23 наконец-то дал простым пользователям инструмент целенаправленного внедрения неопределенного поведения в их код. Такой инструмент, правда, давно уже был и так, но специфичный для конкретного компилятора. C++23 же всего лишь стандартизировал его. Так что радуйтесь, никаких больше уродливых `__builtin_assume`!

"Зачем вообще такая возможность существует?!" — первый же вопрос, который возникает после прочтения абзаца выше. Неужели недостаточно ужасов самого языка, нужно еще пользователям позволить создавать новые?!

На самом деле, конечно, причина есть: компиляторы глупые, быстрый и оптимальный код получить хочется, а на ассемблере писать — не очень. Хотя, конечно, разработчики ffmpeg с этим не согласятся — они поэтому целенаправленно делают ассемблерные вставки, не доверяя компиляторам C.

Несмотря на то что мы говорим о C и C++, я позволю себе привести пример на Rust, поскольку считаю, что он наиболее ярко может продемонстрировать логику нововведения C++23.

Возьмем достаточно простую функцию, которая выполняет семплирование отсортированной выборки: разбивает ее на группы равной величины и из каждой группы выбирает медианную величину:

```
use std::num::NonZeroUsize;

pub fn medians(data: &[f32], group: NonZeroUsize) -> Vec<f32> {
    let n = group.get();
```

```
    data.chunks_exact(n) // разбиваем на группы по n,
                         // последняя группа, если в ней меньше n, игнорируется
        .map(move |chunk| chunk[n/2]) // берем медиану
        .collect() // собираем результат
}
```

Если мы скомпилируем эту функцию довольно старой версией Rustc 1.51 с `opt-level=3`, мы обнаружим, что код получился так себе.

1. Мы видим в начале функции

```
    sub     rsp, 120
    mov     qword ptr [rsp + 96], rcx
    test    rcx, rcx
    je      .LBB4_33
    ...
.LBB4_33:
    ...
    call    qword ptr [rip +
core::panicking::panic_fmt::hcd56f7f635f62c74@GOTPCREL]
    ud2
```

   Это проверка, что `n` не ноль. Но мы же и так знаем, что `n` не ноль: это четко указано в типе входного параметра!

2. При обработке каждой группы мы находим

```
    shr     rdi
    cmp     rdi, r15
    jae     .LBB4_27
    ...
.LBB4_27:
    lea     rdx, [rip + .L__unnamed_4]
    mov     rsi, r15
    call    qword ptr [rip +
core::panicking::panic_bounds_check::h16537cfb53a1364b@GOTPCREL]
```

   Каждый раз проверяется, что индекс `n`/2 в границах группы. Но ведь это всегда так!

Очень бы хотелось донести до компилятора такие очевидные факты. Собственно `[[assume(condition)]]` для того в C++23 и добавили. Если компилятор не смог догадаться до чего-то самостоятельно и сгенерировать оптимальный код, мы теперь можем ему подсказать...

Так, с GCC14 и C++26 та же самая функция (используя безопасные методы, как в Rust)

```
struct NonZero {
public:
    explicit NonZero(size_t v) : value {
```

```
        v > 0 ? v : throw std::runtime_error("Zero value")
    } {}

    size_t get() const {
        return value;
    }
private:
    size_t value;
};

template <class T>
auto chunks_exact(std::span<T> data, size_t n) {
    if (n == 0) {
        throw std::runtime_error("zero chunk len");
    }
    return data.subspan(0, data.size() - data.size() % n)
            | std::views::chunk(n)
            | std::views::transform([](auto chunk){ return std::span(&chunk.front(),
chunk.size()); }); // у элементов chunk view свой особый тип,
                   // сконвертируем их в std::span для консистентности
}

__attribute__((noinline))
std::vector<float> medians(std::span<const float> data, NonZero group) {
    size_t n = group.get();
    // [[assume(n>0)]];
    return chunks_exact(data, n)
            | std::views::transform([n](auto chunk) { return chunk.at(n/2); })
            | std::ranges::to<std::vector>();
}
```

также компилируется со всеми ненужными проверками. Но стоит нам только лишь раскомментировать `[[assume(n>0)]]`, как ситуация меняется, и все избыточные проверки на ноль и на границы групп могут быть успешно выброшены компилятором!

Но что, если мы подсказали неправильно? Неопределенное поведение, конечно же! Из ложной посылки следует что угодно.

А если мы подсказывали правильно, но только на допустимом множестве входных данных? Отлично, всё хорошо, только не забудьте включить в документацию упоминание неопределенного поведения на недопустимом входе.

Но прежде чем начинать пользоваться такой замечательной возможностью языка, стоит понимать:

> Правильная подсказка ничего не гарантирует. А ложная невероятно опасна.

Новые версии компиляторов и сами могут догадаться. Так, например, rustc 1.80 на рассмотренном примере оптимизирует уже всё как надо.

Конструктор по умолчанию и =default

Гайдлайны по современному C++ всячески намекают, а иногда напрямую советуют: следуйте "правилу нуля" (rule of zero)[11] для ваших классов и структур, и будет вам счастье! Используйте инициализаторы по умолчанию! C++20 улучшил поддержку структур-агрегатов, так что не следует писать вручную конструкторы там, где это не надо... Но legacy-код существует, его затратно переписывать... А также существуют legacy-разработчики, которые застряли в C++98...

Так что в старых кодовых базах можно встретить что-нибудь такое:

```
// Point.hpp
class Point2D {
public:
    Point2D(int _x, int _y);
    // Раз добавили какой-то конструктор,
    // нужно добавить и конструктор по умолчанию
    Point();

    int x;
    int y;
};

// Некоторые разработчики как мантру твердят, что
// определение любых функций всегда нужно выносить
// в компилируемый .cpp-файл. Даже коротких.
// Point.cpp
Point2D::Point2D(int _x, int _y) : x {_x}, y {_y} {}
// И даже такие!
Point::Point2D() = default;
```

[11] Rule of zero рекомендует: не определяйте без надобности вручную для своих структур ни деструктор, ни конструкторы/операторы копирования и перемещения. Доверьте это дело компилятору.

Делать так в современном C++ крайне не рекомендуется. Не только из-за обилия бессмысленного бойлерплейта, но и из-за риска получить неинициализированные поля и неопределенное поведение вместе с ними.

Инициализация в C++ — невероятно сложная тема из-за обилия терминологии, переопределенного синтаксиса и вариативности, чтобы удовлетворить все мыслимые и немыслимые возможности. А также из-за множества особых случаев и исключений. И с подобным устаревшим подходом к описанию конструкторов как раз связано одно из таких исключений.

Пусть нам все-таки очень нужно иметь конструкторы для точки.

И мы их определили в составе объявления класса:

```
class Point2D {
public:
  Point2D(int _x, int _y) : x {_x }, y {_y} {}
  // Раз добавили какой-то конструктор,
  // нужно добавить и конструктор по умолчанию
  Point2D() = default;

  int x;
  int y;
};
```

И мы создаем точку, инициализированную по умолчанию с помощью фигурных скобок, как рекомендуется в современном C++:

```
int main() {
  Point2D a {};
  return a.x;
}
```

Стандарт гарантирует, что произойдет zero initialization[12], потому как в классе из тривиальных типов без инициализаторов `Point2D() = default` определил тривиальный конструктор по умолчанию. Так что всё здорово! Никаких неинициализированных полей.

Но стоит нам вынести определение конструктора по умолчанию за пределы объявления класса:

```
class Point2D {
public:
  Point2D(int _x, int _y) : x {_x }, y {_y} {}
  Point2D();
```

[12] Все биты структуры, включая padding, будут заполнены нулями.

```
  int x;
  int y;
};

Point2D::Point2D() = default;
```

как всё резко поменяется! Теперь это уже нетривиальный конструктор. А значит, инициализация фигурными скобками должна вызвать его вместо zero initialization. И поля `x`, `y` останутся неинициализированными. Ведь мы их не инициализировали.

```
struct Bad {
  int x;
  Bad();
};
Bad::Bad() = default;

struct Good {
  int x;
  Good() = default;
};

int main() {
  Bad a {};
  Good b {};
  return a.x + b.x;
}
```

При компиляции GCC с `-std=c++26 -O3 -Wall -Wextra -Wpedantic -Wuninitialized` мы получим предупреждение

```
<source>:15:14: warning: 'a.Bad::x' is used uninitialized [-Wuninitialized]
   15 |     return a.x + b.x;
```

Стоит отметить, что без оптимизаций, ни GCC 14, ни Clang 18 предупреждений не выдают[13].

Ну хорошо. Класс для 2D-точки — это все-таки отличный кандидат, чтобы просто использовать агрегаты и списки инициализации и не думать.

Да. Делайте так!

```
struct Point2D {
  int x = 0;
  int y = 0;
};
```

[13] Некоторые диагностики в LLVM и в GCC привязаны к довольно тяжелому анализу графа потока исполнения программы. Такой анализ необходим для выполнения оптимизаций. Так что диагностики могут не выдаваться, если соответствующие оптимизации не были затребованы.

Я также встречал эту проблему и в более сложных случаях:

Был класс для логирования:

```
class Logger {
public:
    Logger(std::string log_group)
    Logger(); // определен как Logger::Logger() = default в .cpp-файле
private:
    // Это поле было в классе давно. У строк есть конструктор по умолчанию.
    // Инициализатор не обязателен.
    std::string log_group;
};
```

В какой-то момент было решено добавить поле для контроля максимальной длины строки:

```
class Logger {
public:
    Logger(std::string log_group, size_t limit)
    Logger();
private:
    std::string log_group;
    size_t limit; // Неопытный программист, которому поручили задачу,
                  // по аналогии добавил поле без инициализатора
};
```

Всё компилируется, но логгер по умолчанию перестает работать, а `= default` сбивает программиста с толку.

Инициализируйте поля явно! Всегда, кроме случаев, когда инициализация действительно становится проблемой для производительности.

ГЛАВА 4

Стандартная библиотека

NULL-терминированные строки

В начале 1970-х годов Кен Томпсон, Деннис Ритчи и Брайан Керниган, работая над первыми версиями C и UNIX, приняли решение, которое отзывается болью, страданиями, багами и неэффективностью до сих пор, даже спустя 50 лет. Они решили, что строки как данные переменной длины нужно представлять в виде последовательности, заканчивающейся терминирующим символом — нулем. Так делали в ассемблере, а C ведь — высокоуровневый ассемблер! Да и памяти у старенького PDP немного: лучше всего один байтик лишний на строку, чем 2, 4, а то и все 8 байтов для хранения размера в зависимости от платформы... Не-е, лучше байтик в конце! Но в других языках почему-то предпочли хранить размер и ссылку/указатель на данные...

Ну что ж, посмотрим, к чему это привело.

Длина строки

Единственный способ узнать длину NULL-терминированной строки — пройтись по ней и посчитать символы. Это требует линейного времени от длины строки.

```
const char* str = ...;
for (size_t i = 0 ; i < strlen(str); ++i) {
  ...
}
```

И вот уже этот простенький цикл требует не линейного числа операций, а квадратичного. Это азбучный пример. Про него даже известно, что умный компилятор способен вынести вычисление длины строки из цикла:

```
const char* str = ...;
const size_t len = strlen(str);
for (size_t i =0 ; i < len; ++i) {
  ...
}
```

Но ведь пример может быть и посложнее. В коде одной популярной игры про деньги, разборки мафии и угон автомобилей обнаружили занятный пример парсинга большого массива чисел из *JSON*-строки с помощью `sscanf`.

Выглядел он примерно так (его получили путем реверс-инженеринга конечного бинарного файла):

```
const char* config = ...;
size_t N = ...;

for (size_t i = 0; i < N; ++i) {
  int value = 0;
  size_t parsed = sscanf(config, "%d", &value);
  if parsed > 0 {
    config += parsed;
  }
}
```

Прекрасный и замечательный цикл! Тело его выполняется всего *N* раз, но на большинстве версий стандартной библиотеки C каждый раз требуется `strlen(config)` операций на итерацию. Ведь `sscanf` должен посчитать длину строки, чтобы случайно не выйти за ее пределы! А строка NULL-терминированная.

Вычисление длины строки — невероятно часто встречающаяся операция. И один из самых первых кандидатов на оптимизацию — посчитать ее один раз и хранить со строкою... Но зачем тогда NULL-терминатор? Только лишний байт в памяти!

C++ и std::string

C++ — высокоуровневый язык! Уж повыше C, конечно. Стандартные строки в нем, учитывая ошибку C, хранятся как размер плюс указатель на данные. Ура!

Но не совсем "ура". Ведь огромное число библиотек на C никуда не денется, и у большинства из них в интерфейсах используются NULL-терминированные строки. Поэтому `std::string` тоже обязательно NULL-терминированные. Поздравляю, мы храним один лишний байт ради совместимости. А еще мы его храним неявно: `std::string::capacity()` на самом деле всегда на единицу меньше действительно выделенного блока памяти.

C++ и std::string_view

"Используйте `std::string_view` в своих API, и вам не придется писать перегрузки для `const char*` и `const std::string&`, чтобы избежать лишнего копирования!"

Ага, конечно.

`std::string_view` — это тоже указатель плюс длина строки. Но уже, в отличие от `std::string`, указатель не обязательно на NULL-терминированную строку. Ура, мы можем использовать `std::vector` и не хранить лишний байт!

Но если вдруг за фасадом вашего удобного API со `string_view` скрывается обращение к какой-нибудь сишной библиотеке, требующей NULL-терминированную строку...

```
// Эта маленькая программа весело и задорно выведет

// Hello
// Hello World

// Хотите вы этого или нет.

void print_me(std::string_view s) {
  printf("%s\n", s.data());
}

int main() {
  std::string_view hello = "Hello World";
  std::string_view sub = hello.substr(0, 5);
  std::cout << sub << "\n";
  print_me(sub);
}
```

Чуть-чуть изменим аргументы:

```
// Теперь эта маленькая программа весело и задорно выведет

// Hello
// Hello Worldnext (или просто упадет с ошибкой сегментации)

// Хотите вы этого или нет.

void print_me(std::string_view s) {
    printf("%s\n", s.data());
}

int main() {
  char next[] = {'n','e','x','t'};
  char hello[] = {'H','e','l','l','o', ' ',
                  'W','o','r','l','d'};
  std::string_view sub(hello, 5);
  std::cout << sub << "\n";
  print_me(sub);
}
```

Функция не менялась, мы просто передали другие параметры, и всё совсем сломалось! А это всего лишь `print`. С какой-то другой функцией может случиться что-то совершенно немыслимое, когда она пойдет за границы диапазона, заданного в `string_view`.

Что же делать?

Нужно гарантировать NULL-терминированность. А для этого надо скопировать строку... Но ведь `std::string_view` мы же специально использовали в API, чтобы не копировать!

Увы. Как только вы сталкиваетесь со старыми API в C, то, оборачивая их, вы либо вынуждены писать две имплементации — с сырым `char*` и с `const std::string&`, либо соглашаться на копирование на каком-то из уровней.

Как бороться?

Никак.

NULL-терминированные строки — унаследованная неэффективность и возможность для ошибок, от которых мы уже, вероятно, никогда не избавимся. В наших силах лишь постараться не плодить зло: в новых библиотеках на C стараться проектировать API, использующие пару "указатель плюс длина", а не только указатель на NULL-терминированную последовательность.

От этого наследия страдают программы на всех языках, вынужденные взаимодействовать с API в C. Rust, например, использует отдельные типы `CStr` и `CString` для подобных строк, и переход к ним из нормального кода всегда сопровождается очень тяжелыми преобразованиями.

Использование NULL-терминатора встречается не только для текстовых строк. Так, например, библиотека SRILM активно использует нуль-терминированные последовательности числовых идентификаторов, создавая этим дополнительные проблемы. Семейство функций *exec* в Linux принимают NULL-терминированные последовательности указателей. EGL использует для инициализации списки атрибутов, оканчивающиеся нулем. И так далее.

Не нужно дизайнить неудобные, уязвимые к ошибкам API без великой надобности. Экономия в функции на одном параметре размером в указатель редко когда оправдана.

Конструкторы std::shared_ptr

С появления C++11 прошло уже больше 10 лет, так что большинство разработчиков на C++ уже все-таки знают про умные указатели. Отдаешь им владение сырым указателем и спишь спокойно — память будет освобождена. И всё хорошо.

И даже разницу между `std::unique_ptr` и `std::shared_ptr` они знают. Хотя, конечно, у меня был пару лет назад кандидат на собеседовании, который этой разницы не знал, потому что не пользовался STL...

Некопируемый, уникально владеющий std::unique_ptr просто хранит указатель (и, возможно, функцию очистки — deleter), и в своем деструкторе этот deleter применяет к сохраненному указателю. std::shared_ptr же хитрее, и для поддержания разделяемого владения между копиями ему нужен счетчик ссылок. Ну это все знают. Ничего интересного.

Давайте просто пользоваться и не думать.

Удивительно, но на практике в C++ довольно часто встречается ситуация, когда нам нужно описать некую сущность, которую ни в коем случае нельзя создавать на стеке. Она обязательно должна быть в куче.

Простейший пример: нам нужен потокобезопасный объект, который будет внутри защищен мьютексом/атомарными переменными. И мы хотим, чтобы этот объект можно было свободно перемещать из контейнера в контейнер, со стека в контейнер и обратно. А std::mutex и std::atomic конструкторов перемещения не имеют. И у нас два варианта действий в этом случае:

```
class MyComponent1 {
  ComponentData data_;
  // Сделать неперемещаемое поле перемещаемым, добавив
  // к нему слой индирекции и отправив данные в кучу.
  std::unique_ptr<std::mutex> data_mutex_;
};

// Как-то заставить пользователей этого класса создавать
// объекты только на куче и работать с
// std::unique_ptr<MyComponent2> или
// std::shared_ptr<MyComponent2>
class MyComponent2 {
  ComponentData data_;
  std::mutex data_mutex_;
};
```

Второй вариант часто оказывается предпочтительнее, поскольку обращения к мьютексу внутри MyComponent2 обычно происходят чаще, чем загрузки адреса самого объекта. Так что будем раскручивать этот вариант дальше.

Ну и раз мы говорили о потокобезопасном объекте, разумно продолжить с тем, что управлять жизнью нашего объекта мы будем через std::shared_ptr.

Стандартный прием для ограничения создания объектов где попало — сделать конструкторы приватными, а для создания предоставить отдельную функцию.

```
class MyComponent {
public:
  static auto make(Arg1 arg1, Arg2 arg2) ->
    std::shared_ptr<MyComponent>
```

```
  {
    // ???
  }

  // Баним конструкторы копирования и перемещения,
  // чтобы случайно не вытянуть данные объекта в экземпляр на стеке.
  MyComponent(const MyComponent&) = delete;
  MyComponent(MyComponent&&) = delete;
  // И этих друзей тоже баним, но это уже необязательно.
  MyComponent& operator = (const MyComponent&) = delete;
  MyComponent& operator = (MyComponent&&) = delete;

private:
  MyComponent(Arg1, Arg2) { ... };
  ...
};
```

Заглянем внутрь этой фабричной функции `make`. Обычно в этом месте выясняется, что опытный C++-разработчик на самом деле не очень опытный. Но это ему никак не мешает. Да и вообще редко кому мешает.

Можно попробовать написать эту функцию так:

```
auto MyComponent::make(Arg1 arg1, Arg2 arg2) ->
  std::shared_ptr<MyComponent>
{
  return std::make_shared<MyComponent>(std::move(arg1),
                                       std::move(arg2));
}
```

Но нас сразу же ждет разочарование в виде полсотни строк ошибок — `std::make_shared` не может вызвать наш приватный конструктор!

Не беда! И наш C++-разработчик, не сильно напрягаясь, исправляет ошибку:

```
auto MyComponent::make(Arg1 arg1, Arg2 arg2) ->
  std::shared_ptr<MyComponent>
{
  return
    std::shared_ptr<MyComponent>(
      new MyComponent(std::move(arg1), std::move(arg2)));
}
```

Код компилируется, работает. Все свободны?

Действительно, всё работает. Но есть нюанс. Эти два варианта по-разному работают с памятью! Во многих случаях это несущественно. Но если подобным образом создается множество объектов, разница становится заметной.

`std::shared_ptr` считает живые слабые (`weak_ptr`) и сильные ссылки. Для этого ему нужно выделить небольшой блок памяти под пару (атомарных) `size_t` и, может быть, еще что-то. Этот блок зовется *контрольным*.

При использовании `std::make_shared` контрольный блок выделяется рядом с создаваемым объектом, т. е. выделяется один фрагмент памяти как минимум на `sizeof(MyComponent)` + 2 × `sizeof(size_t)`. Это поведение рекомендовано стандартом, но не обязательно к исполнению. Тем не менее все известные имплементации следуют рекомендации.

При вызове конструктора `std::shared_ptr` от сырого указателя объект уже как бы создан и контрольный блок рядом с ним не запихнуть. Поэтому будет выделено еще как минимум 2 × `sizeof(size_t)` памяти. Где-то в другом месте. И тут в ход идут детали реализации аллокаторов, а также пляски с выравниваниями. И в действительности выделяется не `sizeof(MyComponent)` + 2 × `sizeof(size_t)`, а больше. А в случае прямого вызова конструктора от сырого указателя — значительно больше. При расположении контрольного блока рядом с данными также иногда начинают заметно влиять на исход локальность данных и выигрыш от попадания в кеш. Но это всё в случае, если объект маленький.

А если большой?

А если большой, и вы создавали его через `std::make_shared`, а потом плодили `std::weak_ptr`, то у вас может начаться что-то очень похожее на утечку памяти. Хотя объекты исправно умирают, а деструкторы вызываются. Вы же видели это в логе!

Опять-таки: контрольный блок. Если у вас есть живые `weak_ptr`, привязанные к уже отмершим `std::shared_ptr`, то контрольный блок продолжает жить. Ну чтоб вы могли вызвать `std::weak_ptr::expired()`, и он бы вам выдал `true`. Но если контрольный блок сидел в одном куске памяти с умершим объектом, а именно так и получается при создании через `std::make_shared`, кусок памяти из-под объекта операционной системе возвращаться не будет, пока не помрет сам контрольный блок! Вот вам и утечки.

Также есть разница в том, какой именно `operator new` будет вызываться. `std::make_shared` всегда вызывает глобальный. И, если вы перегрузили `new` для своего типа, поведение может быть не таким, какое вы ожидали.

Всё как обычно плохо

Так что же делать, если нам очень надо для своего компонента все-таки выполнить одну аллокацию и потенциально сэкономить? Есть ли решение?

Конечно! В C++ всегда есть какое-нибудь страшное решение. Иногда даже без неопределенного поведения. И это даже наш случай.

Есть `access token` техника, с помощью которой можно осуществить задуманное.

Надо предоставить для `std::make_shared` **публичный** конструктор, но чтобы его можно было вызвать, имея только экземпляр **приватного** типа (*access token*).

```
class MyComponent {
private:
```

```
  // токен доступа
  struct private_ctor_token {
    // только MyComponent может их создавать
    friend class MyComponent;
    private:
      private_ctor_token() = default;
  };
public:
  static auto
  make(Arg1 arg1, Arg2 arg2) -> std::shared_ptr<MyComponent> {
    return
      std::make_shared<MyComponent>(
        private_ctor_token{}, std:: move(arg1), std::move(arg2));
  }

  // Этот конструктор приватный, хотя и в публичной секции.
  // Его никто не сможет вызвать, не имея доступа к приватному токену.
  MyComponent(private_ctor_token, Arg1, Arg2) { ... };
  ...
};
```

И работает.

Сто́ит обратить внимание, что конструктор токена должен быть явно приватным, иначе всю нашу систему безопасности с приватным типом легко обойти вот так:

```
int main() {
  MyComponent c({}, // Создаем приватный токен, не называя его!
                    // У нас нет доступа только к имени.
                {}, {});
}
```

Функция std::shared_from_this

Обсуждая особенности `std::make_shared`, я упомянул, что иногда крайне необходимо убедиться, что объекты вашего класса всегда создаются только в куче и управляются с помощью умного указателя. Вот сейчас будет еще один такой случай.

Вы разрабатываете графический интерфейс и, как это было принято лет 20 назад, решили, что всё должно быть объектно-ориентированно и красиво. Компонентики. Виджеты. Всех мы будем по требованию создавать, управлять умными `shared-` и `weak`-указателями, чтобы не очень сильно задумываться о владении — весьма стандартный подход, между прочим. Rust-библиотеки для GUI, например, часто критикуют за чудовищную сложность именно из-за владений.

Итак, у вас появились некоторые базовые типы:

```
enum class EventType {
  Clicked,
  Created,
  // и другие
};

class Widget {
public:
  virtual ~Widget() = default;
};

class EventListener {
public:
  // Уведомить слушателя, что такой-то виджет породил некоторое событие.
  void notify(EventType, std::weak_ptr<Widget> event_source);
};
```

Хорошо. Дальше давайте заведем кнопку! Куда же без кнопки в хорошем UI?!

```
class Button: public Widget {
public:
  Button(std::shared_ptr<EventListener> listener): listener_ {listener}
  {
    // (1) хотелось бы уведомить слушателя, что кнопка создана!
    listener_->notify(EventType::Created, this); // Это неправильно...
  }

  void click() {
    // (2) хотелось бы уведомить слушателя, что на кнопочку нажали!
    listener_->notify(EventType::Clicked, this); // И это, разумеется, неправильно.
  }

private:
  std::shared_ptr<EventListener> listener_;
};
```

К нашему большому счастью, этот код не компилируется:

```
<source>:26:47: error: cannot convert 'Button*' to 'std::weak_ptr<Widget>'
   26 |         listener_->notify(EventType::Created, this); // Это неправильно...
      |                                               ^~~~
      |                                               |
      |                                               Button*
```

Ах, ну да, типы разные... Опытного разработчика на C++ это, конечно, заставило бы задуматься. А вот неопытного... Он может просто выполнить "преобразование" типов!

```
  Button(std::shared_ptr<EventListener> listener): listener_ {listener}
  {
    // (1) хотелось бы уведомить слушателя, что кнопка создана!
    // Это ОЧЕНЬ неправильно...
    listener_->notify(EventType::Created, std::shared_ptr<Button>(this));
  }

  void click() {
    // (2) хотелось бы уведомить слушателя, что на кнопочку нажали!
    // И это, разумеется, ТОЖЕ неправильно.
    listener_->notify(EventType::Clicked, std::shared_ptr<Button>(this));
  }
```

И взорвется:

```
int main() {
  auto listener = std::make_shared<EventListener>();
  auto button = std::make_shared<Button>(listener);
}

Program returned: 139
free(): invalid pointer
Program terminated with signal: SIGSEGV
```

`std::shared_ptr<Button>(this)` создает новый `shared_ptr`, который ничего знать не знает о существовании другого умного указателя, управляющего объектом. Что, разумеется, приводит к попытке повторного освобождения памяти: сначала одним указателем, затем другим. Что иногда даже может работать успешно... Неопределенное поведение все-таки!

Хорошо, давайте чинить. Забудем пока про конструктор. Попробуем хотя бы починить метод `click`.

Для этого необходимо сообщить кнопке о том, что она управляется умным указателем. Например, вручную добавить в нее `weak_ptr<Button>`-поле и заполнить его после конструирования. Да, это должна быть слабая ссылка, чтобы не создавать цикл из `shared_ptr` и сопутствующую ему утечку памяти:

```
class Button: public Widget {
public:
  // Мы не можем передать weak_ptr<Button> в конструктор!
  // Ведь мы же еще кнопку не создали!
  Button(std::shared_ptr<EventListener> listener): listener_ {listener} {}
```

```
  // Придется сделать что-то такое мерзкое.
  void set_self(std::weak_ptr<Button> self) {
    self_ = self;
  }

  void click() {
    listener_->notify(EventType::Clicked, self_);
  }

  void
private:
  std::shared_ptr<EventListener> listener_;

 std::weak_ptr<Button> self_;
};
```

И пользоваться бы этим добром пришлось так:

```
int main() {
  auto listener = std::make_shared<EventListener>();
  auto button = std::make_shared<Button>(listener);
  button->set_self(button);
  button->click();
}
```

Некрасиво, но работает...

Но C++11 предлагает нам решение получше! `std::enable_shared_from_this`, который позволит нам автоматически добавить такое же `self`-поле и позаботится о его правильном заполнении при конструировании `shared_ptr`.

```
class Button: public Widget, public std::enable_shared_from_this<Button> {
public:
  // Мы не можем передать weak_ptr<Button> в конструктор!
  // Ведь мы же еще кнопку не создали!
  Button(std::shared_ptr<EventListener> listener): listener_ {listener} {}

  void click() {
    listener_->notify(EventType::Clicked, this->weak_from_this());
    // есть также shared_from_this
  }

private:
  std::shared_ptr<EventListener> listener_;
};
```

```
...
int main() {
  auto listener = std::make_shared<EventListener>();
  auto button = std::make_shared<Button>(listener);
  button->click();
}
```

Оно не падает. Красота!

Но что-то всё еще не то. Полная красота не достигнута... Ну разумеется! Ведь в вашем приложении будут не только кнопки, но и комбо-боксы, текстовые поля и прочее... И что, к каждому классу по отдельности `public std::enable_shared_from_this<ClassName>` пририсовывать?

Нет. Разумнее прицепить его к базовому классу и забыть. Давайте так и сделаем.

```
class Widget: public std::enable_shared_from_this<Widget> {
public:
  virtual ~Widget() = default;
};

// Для отладки добавим печать в notify.
class EventListener {
public:
  void notify(EventType, std::weak_ptr<Widget> event_source) {
    std::cout << "event received\n";
    if (auto w = event_source.lock()) {
      std::cout << "widget valid\n";
    }
  }
};
```

Всё работает как надо:

```
Program returned: 0
  event received
  widget valid
```

Прекрасно! Кстати, нам же не очень-то хотелось выпячивать метод `shared_from_this()`, ведь он для внутренних нужд... Может, сделать `protected`-наследование?

```
class Widget: protected std::enable_shared_from_this<Widget> {
public:
  virtual ~Widget() = default;
};
```

И... уже неправильно, но продолжает компилироваться!

```
Program returned: 0
  event received
```

Получили nullptr и рады... Да, ни в коем случае нельзя его наследовать ни приватно, ни защищенно. Так и в документации написано. Настройте себе правило для линтера, пожалуйста.

Кстати, если у вас будет несколько абстрактных интерфейсов, каждый с enable_shared_from_this, и вы попытаетесь унаследовать сразу хоть пару из них... вы получите...

```
class Widget: public std::enable_shared_from_this<Widget> {
public:
  virtual ~Widget() = default;
};

class Gadget: public std::enable_shared_from_this<Gadget> {
public:
  virtual ~Gadget() = default;
};
...
class Button: public Widget, public Gadget {
  ...
  void click() {
    listener_->notify(EventType::Clicked, this->weak_from_this());
  }
};
```

Правильно! Добро пожаловать в ад вариаций ромбовидного наследования!

```
<source>:38:53: error: member 'weak_from_this' found in multiple base classes of
different types
   38 |         listener_->notify(EventType::Clicked, this->weak_from_this());
```

Неоднозначно, откуда метод брать. Но мы можем устранить неоднозначность

```
class Button: public Widget, public Gadget {
  ...
  void click() {
    listener_->notify(EventType::Clicked, static_cast<Widget*>(this)-
>weak_from_this());
  }
};
```

Компилируется. Запускается. Работает неправильно... Опять nullptr.

```
Program returned: 0
  event received
```

Просто не используйте множественное наследование с std::enabled_shared_from_this. И всё будет хорошо.

Ладно. Договоримся, что нас устраивает результат. Мы починили метод click. А как насчет конструктора?

```
class Button: public Widget {
public:
  Button(std::shared_ptr<EventListener> listener): listener_ {listener} {
    listener_->notify(EventType::Created, this->weak_from_this());
  }

  void click() {
    listener_->notify(EventType::Clicked, this->weak_from_this());
  }

private:
  std::shared_ptr<EventListener> listener_;
};

int main() {
  auto listener = std::make_shared<EventListener>();
  auto button = std::make_shared<Button>(listener);
  button->click();
}
```

Работает, но не так, как хочется.

```
Program returned: 0
  event received
  event received
  widget valid
```

Из конструктора мы отправили в Listener нулевой указатель... Ну а чего еще мы хотели?! Ведь работа конструктора еще не завершена. А как мы видели из ручной имитации shared_from_this, внутренний weak_ptr инициализируется только после полного создания объекта. Так что всё работает правильно. Хоть и неожиданно.

Так что если хотите посылать сообщения из конструктора таким образом, то желательно перехотеть. И сделать статический фабричный метод вместо конструктора. Из него уже можно будет посылать уведомления сколько угодно.

```
class Button: public Widget {
public:
  static std::shared_ptr<Button> create(std::shared_ptr<EventListener> listener) {
    auto button = std::make_shared<Button>(listener);
    listener->notify(EventType::Created, button);
    return button;
  }
```

```
  Button(std::shared_ptr<EventListener> listener): listener_ {listener} {}

  void click() {
    listener_->notify(EventType::Clicked, this->weak_from_this());
  }
private:
  std::shared_ptr<EventListener> listener_;
};

int main() {
  auto listener = std::make_shared<EventListener>();
  auto button = Button::create(listener);
  button->click();
}
```

Вот теперь все правильно:

```
Program returned: 0
  event received
  widget valid
  event received
  widget valid
```

Осталось сделать конструктор приватным с помощью приватного тега. Ведь иначе кто-нибудь обязательно создаст кнопку на стеке. И наполучает либо нулевые указатели:

```
int main() {
  auto listener = std::make_shared<EventListener>();
  auto button = Button(listener);
  button.click();
}
Program returned: 0
  event received
```

либо, в случае бесстрашного использования `shared_from_this()` вместо `weak_from_this()`[1], что-то более интересное:

```
class Button: public Widget {
public:
  Button(std::shared_ptr<EventListener> listener): listener_ {listener} {}

  void click() {
    listener_->notify(EventType::Clicked, this->shared_from_this());
  }
```

[1] Который появился только в C++17.

```
private:
  std::shared_ptr<EventListener> listener_;
};

int main() {
  auto listener = std::make_shared<EventListener>();
  auto button = Button(listener);
  button.click();
}

Program returned: 139
terminate called after throwing an instance of 'std::bad_weak_ptr'
  what():  bad_weak_ptr
Program terminated with signal: SIGSEGV
```

Да, сейчас можно получить исключение. А вот до выхода C++17 никакое исключение не гарантировалось. Только неопределенное поведение. Этот дефект исправили и портировали во все версии стандарта, начиная с C++11. Главное, не используйте очень старые версии компиляторов и стандартной библиотеки в их составе.

Потоки ввода-вывода

Стандартная библиотека потоков ввода/вывода в C++ стара, неудобна и полна ужасов. Непосредственно с неопределенным поведением при ее использовании столкнуться проблематично, но нервы потрепать можно. И веселье обычно начинается, когда ваш маленький изолированный и совершенно корректный код, использующий `std::istream` или `std::ostream`, становится частью чего-то большого.

Грабли первые: состояние объекта i/ostream

Даже самым рьяным борцам за чистоту функций и иммутабельность все-таки придется смириться с тем, что где-то глубоко под капотом у сущности, через которую осуществляется ввод/вывод, есть какое-то мутабельное состояние. И это нормально.

Не нормально же то, что у той же самой сущности есть дополнительное мутабельное состояние, ответственное за форматирование данных... Механизм манипуляторов. И он кошмарен.

```
std::cout << std::hex << 10; // 'a', ок
std::cout << 10; // опять 'a' ?!?!
```

Манипулятор меняет состояние потока, переключает режим форматирования для всех последующих операций чтения или записи! И так будет до тех пор, пока не вернут исходное состояние.

```
auto state = std::cout.flags();
std::cout << std::hex << 10; // 'a'
```

```
std::cout.flags(state);
std::cout << 10; // 10, OK
```

Страшно представить, какой хаос начнется, если кто-то передаст в вашу функцию поток с переставленными флагами форматирования. Или вы забудете вернуть их в исходное состояние.

Использование одного и того же имени метода для выставления и получения флагов тоже радует. Особенно любителей возвращать значения через *lvalue*-ссылки в аргументах функций. Но это фишка дизайна чуть ли не всего функционала по настройке потоков. Так что терпим.

Ну и, конечно, состояние форматирования — дополнительная возможность пострелять по ногам в многопоточной среде.

Грабли вторые: глобальная локаль

Мало нам мутабельного состояния с флагами форматирования. Оно хотя бы привязано к конкретному экземпляру `i/ostream`. У нас еще и конструирование новых экземпляров завязано на глобальную мутабельную переменную — текущую глобальную локаль.

Локали — это, конечно, отдельная головная боль. И не только для C и C++, а вообще. Но эта тема выходит за рамки нашей книги.

Здесь же важно лишь то, что `i/ostreams` локалезависимые. И не только они, но и множество функций `std::to_string`, `atof`, `strtol` и прочие прекрасные функции преобразования чего-то к строкам и обратно.

А теперь фокус, демонстрирующий проблему, обнаруживаемую (а потом уныло исправляемую) на каком-то этапе жизни совершенно любой C++-библиотеки, берущейся парсить текстовые форматы данных:

```
int main(int argc, char **argv) {
  auto s = std::to_string(1.5f);
  std::istringstream iss1(s);
  float f1 = 0; iss1 >> f1;
  assert(fabs(f1 - 1.5) < 1e-6); // OK

  std::locale::global(std::locale("de_DE.UTF8"));
  std::istringstream iss2(s);
  float f2 = 0; iss2 >> f2;
  assert(fabs(f2 - 1.5) < 1e-6); // Сюрприз! f2 == 1500000
}
```

Грабли третьи: кодировка путей к файлам и fstream

UTF8 — это прекрасно. UTF8 — это хорошо. У вас код, скорее всего, в UTF8. Python строки по умолчанию гоняет в UTF8. Да все кому не лень гоняют в UTF8! 2025 год. Юникод!

Хотя, конечно, не так уж всё всегда и везде прекрасно. GCC уже 13 лет не могут исправить проблему с BOM-заголовком[2].

А что, если вашей C++-программе приходит UTF8-строка с путем к файлу, который надо открыть?

```
void fun(const std::string& filename) {
  std::ifstream file(filename);
}
```

И всё хорошо? И всё работает? И кириллица? И китайское письмо? Точно работает? А под Windows? И примерно в этот момент выясняется, что все-таки не работает.

Конструктор `std::fstream`, ровно как и сишные `fopen`, особым умом не отличаются. Про ваш Юникод они ничего знать не знают. И что он от нативной кодировки системы внезапно может отличаться, не догадываются.

В итоге мы получаем, что почти каждая C++-программа сталкивается с багом под Windows: сто́ит в пути к файлу возникнуть не-ASCII-символу, так сразу файл не найден.

Грабли четвертые: бинарный режим

Бинарный режим (binary mode) чтения и записи файлов — еще одна отдельная боль, от которой страдают на самых разных языках. Чтение бинарных данных из `stdin`, запись в `stdout` (которые по умолчанию открыты в текстовом режиме), теряющиеся или лишние добавляемые байты CR (`\r`) — всё как мы любим.

Но в C++ у нас есть дополнительные возможности для страданий.

Я довольно часто встречаю эту ошибку, и не только в студенческих работах:

```
std::ifstream file(name, std::ios::binary);
char x = 0;
file >> x; // Считают, что будет чтение одного байта.
```

Но нет: `operator>>` для стандартных типов всегда пытается выполнить форматное чтение, и по умолчанию все пробельные символы будут пропущены. Более того, у нас в принципе отсутствует возможность узнать, в каком режиме открыт поток! Нужно самостоятельно где-то сохранить информацию об этом.

Аналогично, но ошибка быстрее проявляется:

```
std::ifstream file(name, std::ios::binary);
int x = 0;
file >> x; // Считают, что будет чтение sizeof(int) байт.
```

Также очень распространен особо мерзкий случай:

```
std::ifstream file(name, std::ios::binary);
std::string s;
file.read(reinterpter_cast<char*>(&s), sizeof(s)); // Неопределенное поведение!
```

[2] При наличии `Byte-order-marker` в начале заголовочного файла (.h/.hpp) GCC игнорирует директивы `#pragma once`. И это отлично ломает сборку!

Неопытные программисты, тестирующие на коротких строках и успокаивающиеся на этом, могут столкнуться с тем, что код будет работать "так, как они и предполагали". Исключительно из-за особенности современной реализации строк и техники SSO (*small string optimization*): строка реализуется не просто как три поля (`data`, `size`, `capacity`), а, если она короткая, записывается прямо поверх этих полей.

Но, конечно же, это некорректно.

Грабли пятые: ошибки чтения и конец потока

У потоков ввода/вывода есть еще флаги — состояние потока: были ли ошибки, закончились ли данные. И многие знают, что проверить успешность операции можно, засунув объект потока в условный оператор (или в иной другой контекст, где выполняется приведение к `bool`).

А те, кто не знает, используют проверку вида `while (!iss.eof())` и однажды наступят на грабли вечного цикла. Такое произойдет, когда файл еще не закончился, но и читать его дальше нельзя. Например, если это файл на сетевом диске, и сеть отвалилась. Впрочем, это уже совсем другая история. Вернемся к корректной проверке возможности чтения.

```
std::istringstream iss("\t aaaa \n bb  \t ccc dd e ");
std::string token;
int count = 0;
while (iss >> token) {
  ++count;
}
assert(count == 5); // OK
```

Тут будут прочитаны все пять токенов из строки. Ни больше, ни меньше.

Если будет ошибка:

```
std::istringstream iss("1 2 3 pr 5");
int token = 0;
int count = 0;
while (iss >> token) {
  ++count;
}
std::cout << token; // Выведет 0!
assert(count == 3); // OK
```

то всё логично. А на токене, на котором произошла ошибка, результат зануляется. Если сильно надо, можно настроить выбрасывание исключений.

А что, если бинарные данные почитать?

```
std::istringstream iss("12345");
std::array<char, 4> buf;
int read_count = 0;
```

```
while (iss.read(buf.data(), 4)) {
  read_count += iss.gcount();
}
assert(read_count == 5); // Упс, последний байт не учелся.
```

А тут у нас EOF при чтении. А значит, ошибка. И все равно, что один байт-то прочитался успешно.

Ну хорошо, в C есть прекрасные `fread`, которые сразу возвращают количество считанных байтов, и получается красивый цикл. Может, что-то такое есть и у C++-потоков? Конечно есть!

Грабли шестые: readsome

```
std::istringstream iss("12345");
std::array<char, 4> buf;
int read_count = 0;
while (iss.readsome(buf.data(), 4) > 0) {
  read_count += iss.gcount();
}
assert(read_count == 5);
```

Вау, работает!

На самом деле, нет. Идем на cppreference и читаем:

> The behavior of this function is highly implementation-specific. For example, when used with `std::ifstream`, some library implementations fill the underlying `filebuf` with data as soon as the file is opened (and `readsome()` on such implementations reads data, potentially, but not necessarily, the entire file), while other implementations only read from file when an actual input operation is requested (and `readsome()` issued after file opening never extracts any characters).

Что в переводе означает:

> Поведение этой функции в значительной степени зависит от реализации. Например, при использовании с `std::ifstream`, некоторые реализации библиотеки заполняют базовый `filebuf` данными, как только файл открывается (и `readsome()` в таких реализациях читает данные, потенциально, но не обязательно, весь файл), в то время как другие реализации читают из файла только тогда, когда запрашивается фактическая операция ввода (и `readsome()`, вызванный после открытия файла, никогда не извлекает никаких символов).

В общем, не работает. Упражнение по замене `istringstream` на `ifstream` в примере выше предлагаю читателю проделать самостоятельно.

Функции стандартной библиотеки как параметры

Предположим, вам нужно заниматься какими-то вычислениями по долгу службы (или вы просто студент, и вам кровь из носу надо выполнить задание по численным методам).

И вы взяли готовую прекрасную функцию интегрирования:

```
template <class F>
concept NumericFunction =
  std::is_invocable_v<F, float> && requires (float arg, F f) {
    { f(arg) } -> std::convertible_to<float>;
};

float integrate(NumericFunction auto f) {
  float sum = 0;
  /* Мы не будем вдаваться в подробности точности,
     сходимости, шагов разбиения и выбора точки,
     хотя это тоже очень важно, но в другой книжке (по численным методам). */
  for (float x : std::views::iota(1, 26)) {
    sum += f(x);
  }
  return sum;
}
```

Отлично! Вы начинаете ее тестировать на какой-нибудь стандартной функции:

```
#include <cmath>
...
int main() {
  return integrate(sqrt); // Все OK?
                          // (Приведение к int считаем нормальным.)
}
```

Вроде, да. Программа возвращает `85`.

На самом деле нет. И проблемы тут как минимум две.

1. Стандартная библиотека C++ содержит в себе стандартную библиотеку C, что делает использование стандартных математических функций особенно болезненным:

   ```
   static_assert(std::abs(5.8) > 5.5);
   static_assert(abs(5.8) > 5.5);
   ```

```
//--------------------

<source>:26:24: error: static assertion failed
   26 | static_assert(abs(5.8) > 5.5);
      |               ~~~~~~~~~^~~~
<source>:26:24: note: the comparison reduces
to '(5.0e+0 > 5.5e+0)'
Compiler returned: 1
```

О'кей. Мы поняли. Надо использовать `std::sqrt`, чтобы не попасть не в ту перегрузку.

```
int main() {
  return integrate(std::sqrt);
}

// --------------

<source>:22:21: error: no matching function for call to
'integrate(<unresolved overloaded function type>)'
   22 |     return integrate(std::sqrt);
```

Ага, `overloaded function type`. И как же нам тогда выбрать нужный?

Вы пошли в Google с этим вопросом, и первая ссылка привела вас на какой-нибудь Qt-форум[3]. И самый релевантный ответ был — соорудить явное приведение типов указателей на функцию.

```
int main() {
  return integrate(
    static_cast<float(*)(float)>(&std::sqrt));
}
```

Ура, работает! Программа по-прежнему возвращает `85`. Но это немного по-другому посчитанные `85`.

Поздравляю...

2. Вы нарушили пункт 16.4.5.2.6:

> "Let F denote a standard library function ([global.functions]), a standard library static member function, or an instantiation of a standard library function template. Unless F is designated an addressable function, the behavior of a C++ program is unspecified (possibly ill-formed) if it explicitly or implicitly attempts to form a pointer to F".

[3] О, в Qt это распространенная проблема — указать перегрузку при соединении сигналов и слотов.

Вызов `integrate(static_cast<float(*)(float)>(&std::sqrt));` делает именно это. Вы взяли указатель на функцию. Указатели почти на любую функцию стандартной библиотеки **брать нельзя**.

Изначальный вариант с `return integrate(sqrt)`, использующий `sqrt` из библиотеки C, также попадает в эту ловушку, только неявно. А с версии C++20 грозят, что может перестать компилироваться, но я пока такого не видел.

Почему же нельзя?

А кто вам сказал, что это функция?

Да, почти все функции стандартной библиотеки C++17 после инстанциирования шаблонов все-таки оказываются нормальными функциями, и потому у нас уж сколько лет всё работает.

С функциями стандартной библиотеки C всё, конечно, хуже — они могут быть макросами, и черт его знает от чего вы на самом деле взяли адрес в таком случае.

С версии C++20 (вдохновленные *`ranges`* Эрика Ниблера) новые[4] функции внезапно могут оказаться *ниблоидами*. Глобальными объектами с определенным `operator()`. Так что они могут выглядеть и крякать как старые добрые функции, но таковыми не быть. И, если вы использовали *C-style*-каст вместо громоздкого `static_cast`, вас могут ждать интересные результаты:

```
// Старая версия
// float f(float a) {
//   return a;
// }

// Вы обновились, и теперь это ниблоид!
auto f = [](float a) -> float {
  return a;
};

int main() {
  return integrate((float(*)(float))(&f));
  // Ошибка сегментации.
}
```

Положение могло бы спасти отсутствие `&` перед именем функции. Для функций и лямбд применяется неявный `decay` к указателю:

```
// float f(float a) {
//   return a;
// }

auto f = [](float a) -> float {
  return a;
};
```

[4] А также потенциально старые, после перехода `std` на модули.

```
int main() {
  return integrate((float(*)(float))(f));
  // Компилируется и работает.
}
```

Ниблоиды в std чаще определяются так, как показано ниже, а не с помощью лямбд:

```
struct {
  static float operator()(float x) {
    return x;
  }
} f;

int main() {
  return integrate((float(*)(float))(f));
  // Ошибка: недопустимое приведение от типа '<unnamed struct>'
  // к типу 'float (*)(float)'.
}
```

Не компилируется, и нам невероятно повезло, что это так!

Но, к сожалению, примеров кода с явным взятием адреса функции очень много в мире.

Что же делать?

Хорошая новость: если когда-нибудь вся замечательная гора стандартных функций станет вызываемыми объектами, ваш integrate(std::sqrt) будет компилироваться и работать правильно из коробки. И все будут счастливы.

Плохая новость: замечательно не будет, поэтому придется писать код.

Проблема решится оборачиванием вызова к std функции в вашу функцию или в лямбду.

```
int main() {
  return integrate([](float x) {
    return std::sqrt(x);
  });
}
```

Или можно завести вспомогательный макрос. Если использовать C++20, то он выглядит менее страшно, чем обычно:

```
#define LAMBDA_WRAP(f) []<class... T>(T&&... args) \
  noexcept(noexcept(f(std::forward<T>(args)...))) -> \
    decltype(auto) \
  { return f(std::forward<T>(args)...); }

int main() {
  return integrate(LAMBDA_WRAP(std::sqrt));
}
```

Причем вариант с лямбдой, а не с функцией будет почти всегда предпочтительнее по соображениям оптимизаций. Смотрите.

Если вызов шаблонной функции `integrate` по какой-либо причине не может быть заинлайнен компилятором и вы передаете указатель на функцию, компилятор не имеет никакого выбора, кроме как генерировать инструкцию `call` по этому указателю:

```
#define LAMBDA_WRAP(f) []<class... T>(T&&... args) \
  noexcept(noexcept(f(std::forward<T>(args)...))) -> \
    decltype(auto) \
  { return f(std::forward<T>(args)...); }

float my_sqrt(float f) {
  return std::sqrt(f);
}

int main() {
  return integrate(my_sqrt) + integrate(LAMBDA_WRAP(std::sqrt));
}
```

Ассемблерный код при использовании функции:

```
// float integrate<float (*)(float)>(float (*)(float)):
    push    rbp
    mov     rbp, rdi
    ...
.L24:
    pxor    xmm0, xmm0
    cvtsi2ss        xmm0, ebx
    add     ebx, 1
    call    rbp  // ! нет информации о функции - вызов по указателю
    addss   xmm0, DWORD PTR [rsp+12]
    movss   DWORD PTR [rsp+12], xmm0
    cmp     ebx, 26
    ...
    ret
```

Ассемблерный код при использовании лямбды:

```
float integrate<
  main::{lambda<typename... $T0>(($T0&&)...)#1}>(
    main::{lambda<typename... $T0>(($T0&&)...)#1})
      [clone .isra.0]:
    ...
```

```
.L16:
    pxor    xmm0, xmm0
    cvtsi2ss        xmm0, ebx
    ucomiss xmm2, xmm0
    ja      .L21
    sqrtss  xmm0, xmm0 // ! sqrt подставлен
    add     ebx, 1
    addss   xmm1, xmm0
    cmp     ebx, 26
    jne     .L16
.L11:
    ...
.L21:
    movss   DWORD PTR [rsp+12], xmm1
    add     ebx, 1
    call    sqrtf  /// ! sqrt подставлен
    ...
```

Почему лямбда предпочтительнее, но не всегда?

GCC и Clang, например, генерируют копию кода для каждого вызова с лямбдой, даже если они одинаковые. Ну просто потому, что так необходимо: у каждой лямбды должен быть уникальный тип.

```
int main() {
  return integrate(my_sqrt) +
         integrate(LAMBDA_WRAP(std::sqrt)) +
         integrate(LAMBDA_WRAP(std::sqrt)) +
         integrate(LAMBDA_WRAP(std::sqrt));
}
```

Что поделать, раздутие кода — известный результат мономорфизации[5] шаблонов/generic-функций.

Используйте лямбду повторно, и будет лучше:

```
// Сгенерированный код в два раза меньше, чем для примера выше.
int main() {
  auto sqrt_f = LAMBDA_WRAP(std::sqrt);
  return integrate(my_sqrt) +
         integrate(sqrt_f) +
         integrate(sqrt_f) +
         integrate(sqrt_f);
}
```

[5] Простите за страшное слово для процесса инстанцирования шаблонов после фиксации их параметров.

Представление std::ranges::views

Как-то лениво…

2025 год. C++20 уже как пять лет готов[6] к использованию в серьезной production-разработке. По крайней мере, мне недавно сообщили, что компиляторы наконец-то обновлены, и мы уже можем...

C++20 принес четыре крупные фичи. Две из них сразу готовы к употреблению в вашем коде, а еще две не очень. Здесь мы будем говорить о первых двух.

Библиотека std::ranges

Революция в методе работы с последовательностями элементов в C++! Последний раз такое было, когда в C++11 `range-based-for` сделали. И вот опять.

Забудьте о паре итераторов `begin`/`end` и мучениях с тем, чтобы лихо и изящно выбросить все нечетные числа, а все четные возвести в квадрат, как это можно сделать в других высокоуровневых языках:

```
let v : Vec<_> = ints.iter()
                     .filter(|x| x % 2 == 0)
                     .map(|x| x * x)
                     .collect();
```

```
List<int> v = Stream.of(ints)
      .filter(x -> x % 2 == 0)
      .map(x -> x * x)
      .collect(Collectors.toList());
```

```
var v = ints.Where(x => x % 2 == 0)
            .Select(x => x * 2)
            .ToList();
```

```
// До C++20
std::vector<int> v;
std::copy_if(ints.begin(), ints.end(),
             std::back_inserter(v), [](int x)
             { return x % 2 == 0;});
std::transform(v.begin(), v.end(), v.begin(),
               [](int x){return x * x;});
```

[6] Не совсем. Включение модулей всё еще вызывает проблемы и по умолчанию плохо поддерживается.

```
// После C++20
std::vector<int> v;
std::ranges::copy(
    ints | std::views::filter([](int x){ return x % 2 == 0;})
         | std::views::transform([](int x) { return x * x;})
    std::back_inserter(v)
);

// После C++23
auto v =
    ints | std::views::filter([](int x){ return x % 2 == 0;})
         | std::views::transform([](int x) { return x * x;})
         | std::ranges::to<std::vector>();
```

Красота! Только компилируется оно долго и оптимизируется не всегда хорошо, но ничего страшного...

Концепты

Концепты (concepts) были нужны и как самостоятельная фича: слишком SFINAE в современном C++ необходим то здесь, то там (особенно тем, кто пишет библиотеки), а техника эта тяжелая и для чтения, и для написания. И ошибки компиляции чудовищные... Концепты как именованные ограничения должны были улучшить ситуацию.

Так что теперь мы, например, можем написать максимально правильную и, вероятно, понятную generic-функцию для сложения целых чисел. И только целых. И это будет явно видно в ее сигнатуре безо всякой магии `enable_if`.

```
std::integral auto sum(std::integral auto a
                       std::integral auto b) {
  return a + b;
}
```

И для `ranges` они были страшно необходимы. Вся библиотека `ranges` — это заоблачная гора шаблонов с ограничениями, и без синтаксического сахара концептов прочитать сигнатуры функций было бы невероятно сложно, а в попытках разрешить ошибки компиляции можно было бы провести вечность.

А теперь стреляем с двух рук!

Один программист написал в общий C++-чат, что он работает над добавлением тестов к какой-то новой фиче в библиотеке, которая так здорово реализована с помощью C++20 и `ranges`, да вот только что-то странное происходит: тесты падают, Valgrind что-то совершенно невразумительное говорит...

Код фичи был таким:

```
struct MetricsRecord {
   std::string tag;
   // ...
};

struct Metrics {
  std::vector<MetricsRecord> records;

  std::ranges::range auto by_tag(const std::string& tag) const;
  // ...
};

// ... много-много кода

std::ranges::range auto
Metrics::by_tag(const std::string& tag) const {
  return records |
         std::ranges::views::filter([&](auto&& r)
           { return r.tag == tag; });
}
```

Ничего страшного, вроде все нормально. Никаких проблем не видно.

Но посмотрим тесты:

```
int main() {
  auto m = Metrics {
    {
      {"hello"}, {"world"}, {"looooooooooooooongtag"}
    }
  };

  {
    // печатает found
    auto found = m.by_tag("hello");
    for (const auto& f: found) {
      std::cout << std::format("found {}\n", f.tag);
    }
  }

  {
    // не печатает... странно
    auto found = m.by_tag("looooooooooooooongtag");
```

```
    for (const auto& f: found) {
      std::cout << std::format("found {}\n", f.tag);
    }
  }

  {
    // а так работает
    std::string tag = "looooooooooooooongtag";
    auto found = m.by_tag(tag);
    for (const auto& f: found) {
      std::cout << std::format("found {}\n", f.tag);
    }
  }
}
```

Искушенный читатель уже понял, что проблема во временной переменной:

```
// Неявное создание временной переменной
// std::string в аргументе!
auto found = m.by_tag("looooooooooooooongtag");
```

и в том, что предикат фильтрации захватывает переменную по ссылке:

```
std::ranges::range auto
Metrics::by_tag(const std::string& tag) const {
  return records |
         std::ranges::views::filter([&](auto&& r)
           { return r.tag == tag; });
}
```

А еще в том, что разработчик долгое время писал на JavaScript, а там `Array.prototype.filter` сразу же создает новый массив:

```
const words = ['spray', 'elite', 'exuberant',
               'destruction', 'present'];

const result = words.filter((word) => {
   console.log(word);
   return word.length > 6
  });
// Сразу же будут напечатаны все элементы.
```

Разработчик просто не понял сразу, что метод ленивый (и что все `std::ranges` ленивые).

Все так просто! Вам нужно лишь правильно использовать C++, и проблем не будет! Или будут?..

```
std::ranges::range auto
  Metrics::by_tag(const std::string& tag) const
```

Можно ли по сигнатуре метода догадаться, что он ленивый? Вряд ли. `std::ranges::range auto` не дает об этом никакой информации.

Это должно быть написано в документирующем комментарии. Но его не было. Либо должен был быть использован концепт `std::ranges::view`. Ах, как обычно, вот если бы всё было сделано правильно...

Но зато Valgrind поймал ошибку! Да. В тестах к библиотеке. Кто знает, есть ли тесты у пользователей этой библиотеки...

Пусть пишут тесты! Пусть используют статический анализ! Ага. Возможно, они помогут. **На момент написания этого текста (апрель 2024 года)**: ни Clang-Tidy, ни PVS-Studio не могут диагностировать эту ошибку.

Ну хорошо. Все теперь начнут заранее читать документацию и будут знать, что `std::viewes` — ленивые. А значит, захватывать ссылки с ними нужно осторожно. Закрываем вопрос.

Подождите[7]. `std::ranges` не просто ленивые, а невероятно ленивые! Им иногда не просто лень итерироваться по контейнеру, а даже лень у контейнера `begin()` и `end()` лишний раз вызвать. Причина такой лени — требования стандарта обеспечить в среднем константное время выполнения методов `begin()` и `end()`:

> Given an expression t such that decltype((t)) is T&, T models range only if (2.1)
>
> [ranges::begin(t), ranges::end(t)) denotes a range ([iterator.requirements.general]), (2.2) both ranges::begin(t) and ranges::end(t) are amortized constant time and non-modifying, and ...

Поэтому некоторые `views`:

- откладывают вызов `begin`/`end` у контейнера при конструировании;
- кешируют свои `begin`/`end` после их первого вычисления.

И получаются интересные спецэффекты:

```
void print_range(std::ranges::range auto&& r) {
  for (auto&& x: r) {
    std::cout << x << " ";
  }
  std::cout << "\n";
}

void test_drop_print() {
  std::list<int> ints = {1, 2, 3 ,4, 5};
```

[7] Изложенное далее во многом вдохновлено выступлением Nicolai Josuttis (**https://twitter.com/NicoJosuttis**) на конференции "Keynote Meeting C++ 2022" (**https://www.youtube.com/watch?v=O8HndvYNvQ4**).

```
  auto v = ints | std::views::drop(2); // Пропустить первые два.
  ints.push_front(-5);
  print_range(v); // -5 и 1 пропущены.
                  // drop вызвал begin и end только сейчас.
}

void test_drop_print_before_after() {
  std::list<int> ints = {1, 2, 3 ,4, 5};
  auto v = ints | std::views::drop(2);
  print_range(v); // 1, 2 пропущены
  ints.push_front(-5);
  print_range(v); // 1, 2 пропущены!
                  // drop не вызывает begin и end еще раз.
}

void test_take_print() {
  std::list<int> ints = {1, 2, 3 ,4, 5};
  auto v = ints | std::views::take(2);
  ints.push_front(-5);
  print_range(v); // -5, 1 выведены
}

void test_take_print_before_after() {
  std::list<int> ints = {1, 2, 3 ,4, 5};
  auto v = ints | std::views::take(2);
  print_range(v); // 1, 2 выведены
  ints.push_front(-5);
  print_range(v); // -5, 1 выведены!
                  // take вызывает begin и end каждый раз.
}
```

Будет выведено:

```
drop:
2 3 4 5
------
3 4 5
3 4 5
take:
-5 1
------
1 2
-5 1
```

Здорово! Совершенно естественно, а главное — предсказуемо! Нет никакой магии, если знать как оно работает... Главное не ошибиться при использовании на практике.

Просто не надо брать и модифицировать контейнер, когда на него взят `ranges::view`. Это же так просто.

Кстати, если мы сделаем одно крохотное изменение:

```
void print_range(std::ranges::range auto r) // Теперь по значению.
{
  for (auto&& x: r) {
    std::cout << x << " ";
  }
  std::cout << "\n";
}

void test_drop_print_before_after() {
  std::list<int> ints = {1, 2, 3 ,4, 5};
  auto v = ints | std::views::drop(2);
  print_range(v); // 1, 2 пропущены.
  ints.push_front(-5);
  print_range(v); // -5, 1 пропущены!
                  // Мы же теперь сделали копию view, и
                  // копия снова вызвала begin() и end().
}
```

следующим вытекающим спецэффектом такого ленивого и иногда кеширующего поведения является то, что в функцию, принимающую `const std::range::range&`, абы какой `view` подставить нельзя.

```
void print_range(const std::ranges::range auto& r) {
  for (auto&& x: r) {
    std::cout << x << " ";
  }
  std::cout << "\n";
}

void test_drop_print() {
  std::list<int> ints = {1, 2, 3 ,4, 5};
  auto v = ints | std::views::drop(2);
  print_range(v); // Не компилируется!
                  // drop от std::list должен быть мутабельным.
  /*
    <source>: In instantiation of
      'void print_range(const auto:42&) [with auto:42 =
      std::ranges::drop_view<std::ranges::ref_view<
        std::__cxx11::list<int> > >]':
```

```
    <source>:19:16:   required from here
             19 |     print_range(v);
                |     ~~~~~~~~~~~^~~
    <source>:10:5: error: passing 'const std::ranges::drop_view<
      std::ranges::ref_view<std::__cxx11::list<int> > >' as
      'this' argument discards qualifiers [-fpermissive]
             10 |     for (auto&& x: r) {
  */
}

void test_drop_print_vector() {
  std::vector<int> ints = {1, 2, 3 ,4, 5};
  auto v = ints | std::views::drop(2);
  print_range(v); // Все OK.
}
```

Соответственно, один и тот же абстрактный `view` ни в коем случае нельзя напрямую использовать по ссылке в нескольких потоках. Нужно требовать константности или синхронизировать доступ. Также писатели generic-кода должны возложить на себя дополнительную когнитивную нагрузку и правильно указывать концепты-ограничения.

Для начала вот эти четыре:

1. `std::ranges::range` — слишком абстрактный: только *`begin`* и *`end`*.
2. `std::ranges::view` — тоже *`range`*, но ему удовлетворяют только *`views`*.
3. `std::ranges::borrowed_range` — тоже слишком абстрактный, но его итераторы безопасно возвращать из функций.
4. `std::ranges::constant_range` (C++23) — тоже абстрактный, но итераторы дают только read-only-доступ.

А потом еще и оставшиеся подключатся.

Последним выдающимся спецэффектом ленивого кеширования является следующий курьез:

```
enum State {
  Stopped,
  Working,
  ...
};

struct Unit {
  State state;
  ...
};
```

```
...

std::vector<Unit> units;
...

// Остановить все работающие единицы.
for (auto& unit: units | std::views::filter{[](auto& unit)
  { return unit.state == Working; }}) {
    ...
    unit.state = State::Stopped; // Неопределенное поведение!
    // https://eel.is/c++draft/range.filter#iterator-1
    /*
       Допустимо изменять элемент, указанный filter_view::iterator, но
       это приведет к неопределенному поведению,
       если результирующее значение не удовлетворяет предикату фильтра.
    */
}
```

Стандарт явно запрещает изменять элементы, найденные с помощью `std::views::filter`, таким образом, чтобы результат предиката менялся! Всё из-за предположения, что вы, возможно, еще раз будете итерироваться по тому же самому `view`. И, чтобы не делать работу дважды, нужно закешировать `begin()`.

Самое чудовищное, что такое поведение закреплено стандартом[8]. Это не определено в реализации:

> Для обеспечения амортизированной константной сложности по времени, которая требуется в соответствии с концепцией `range`, когда `filter_view` моделирует `forward_range`, эта функция кеширует результат в `filter_view` для использования при последующих вызовах.

Конструкция std::views::transform | filter

zero-cost-абстракция,
за которую нужно платить дважды.

Когда C++20 анонсировал добавление `ranges` в стандартную библиотеку, было очень много обсуждений. Кто-то радовался (как, например, я, по наивности), кто-то,

[8] См. **https://eel.is/c++draft/range.filter#view-5**.

наоборот, высказывал опасения насчет этой новой и якобы удобной функциональности. Особенно много возмущений и критики было со стороны разработчиков игр:

- эти шаблоны на шаблонах компилируются долго (и это правда);
- старый добрый `for loop` читается проще (это сомнительно);
- эти абстракции дают еще и штраф в производительности (а вот это мы сейчас и обсудим).

Пример с бенчмарком `transform | filter` против обычного старого дедовского `for loop` обсуждался на форумах, показывался на конференциях, ну и здесь тоже без него не обойдется.

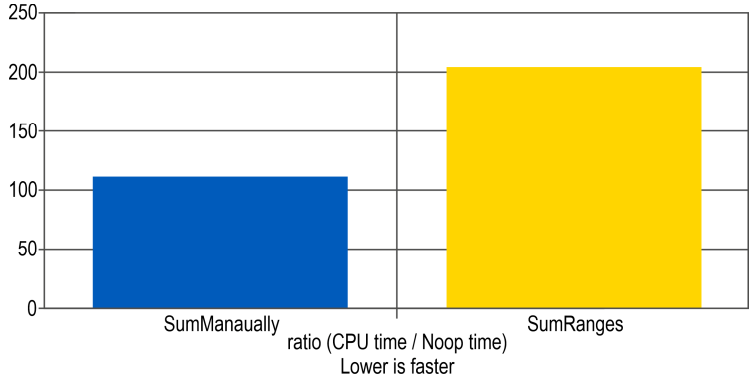
Как известно, `std::ranges` придумали, чтобы в C++ было удобнее реализовывать печать календаря[9], а также суммировать квадраты чисел. Календарь печатать слишком долго, поэтому будем суммировать квадраты.

```
// Суммировать будем "маленькие" квадраты,
// чтобы как-то оправдать использование filter после transform.

// Будем суммировать вручную, старым проверенным способом.
static int sum_small_squares_manualy(std::vector<int>& v) {
  int sum = 0;
  for (int x : v) {
      int sq = x * x;
      if (sq < 100) {
        sum += sq;
      }
  }
  return sum;
}

// А тут просуммируем, используя почти всю мощь современного C++: ranges::views!
static int sum_small_squares_ranges(std::vector<int>& v) {
  namespace views = std::ranges::views;
  int sum = 0;
  auto square = [](int x) { return x * x; };
  auto small = [](int x) { return x < 100; };
  for (int x : v | views::transform(square)
                 | views::filter(small)) {
      sum += x;
  }
  return sum;
}
```

[9] В 2015 году Эрик Ниблер представил свою библиотеку `ranges` на конференции "Northwest C++ Users Group". В качестве мотивирующего примера он демонстрировал задачу печати календаря.

И запустим бенчмарк. Результат его, скорее всего, известен моему читателю. (Ну правда, его везде показывали!).

Версия `std::ranges` почти ровно в 2 раза медленнее, чем традиционная реализация:



Что для libc++, что для libstdc++, что для Clang, что для GCC — всё едино.

Самое страшное, что это не какая-то упущенная оптимизация компиляторов. Это так и задумано.

В этой книге я обычно освещаю случаи undefined- и unspecified-поведения. И очень редко какие-то well-defined-вещи. Однако ситуация с `transform | filter`-сочетанием оказалась настолько неожиданной, что обойти ее стороной нельзя. И речь не о производительности.

Давайте рассмотрим другой пример, более приближенный к реальности. Ведь на C++ не только квадраты вычисляют, но еще и какую-то бизнес-логику программируют.

Вы подготовили список API calls-запросов к AWS и собираетесь их исполнить; какие-то могут завершиться ошибкой, какие-то выполниться успешно.

Конструкция `requests | transform(execute) | filter(succeded)` выглядит очень чисто, красиво и понятно.

Вы пишете ее, используете, тестируете — вроде всё отлично работает. Загоняете в продакшен... И потом получаете удвоенный счет от AWS.

Что? А давайте посмотрим:

```
using Response = std::string;
using Request = int;
using Error = int;

auto execute_request(Request r) -> std::expected<Response, Error> {
    std::print("Executing request {}\n", r);
    if (r % 2 == 0) {
        return std::format("{}", r);
```

```
    } else {
        return std::unexpected(r);
    }
}

int main() {
    namespace views = std::ranges::views;
    std::vector<Request> requests = { 1, 2, 3, 4, 5, 6 };
    constexpr auto successful = [](auto&& response) { return response.has_value(); };
    for (auto resp : requests | views::transform(execute_request)
                              | views::filter(successful)) {
        std::print("got result: {}\n", *resp);
    }
}
Executing request 1
Executing request 2
Executing request 2
got result: 2
Executing request 3
Executing request 4
Executing request 4
got result: 4
Executing request 5
Executing request 6
Executing request 6
got result: 6
```

Глядите-ка! Все успешные запросы выполнялись дважды!..

Ох, а вдруг в вашем коде была какая-то сложная логика фильтрации, и вам нужно было сначала отфильтровать успешные запросы, а потом отфильтровать их еще по какому-то критерию...

```
for (auto resp : requests | views::transform(execute_request)
                          | views::filter(successful)
                          | views::filter(other_condition)) {
    std::print("got result: {}\n", *resp);
}
```

И вот они уже исполняются трижды!

```
Executing request 1
Executing request 2
Executing request 2
Executing request 2
got result: 2
```

```
Executing request 3
Executing request 4
Executing request 4
Executing request 4
got result: 4
Executing request 5
Executing request 6
Executing request 6
Executing request 6
got result: 6
```

Но ведь это же не всё! Если запрос исполняется повторно, то при повторном исполнении он может завершиться неуспешно. И внезапно в этом случае мы вполне можем получить UB с разыменованием `nullptr`/`nullopt`/`unexpected`.

```
// Давайте смоделируем такую возможность с помощью генератора случайных
// чисел [мы не вызывали srand, так что поведение будет стабильным].
auto execute_request(Request r) -> std::expected<Response, Error> {
    std::print("Executing request {}\n", r);
    if (std::rand() % 2 == 0) {
        return std::format("{}", r);
    } else {
        return std::unexpected(r);
    }
}
...

for (auto resp : requests | views::transform(execute_request)
                          | views::filter(successful)) {
    // И вот тут мы разыменуем ошибочный std::expected несмотря на то,
    // что мы "проверили" его в filter.
    std::print("got result: {}\n", *resp);
}
```

Бах!

```
Execution build compiler returned: 0
Program returned: 139
Program terminated with signal: SIGSEGV
```

Эх, получается, все-таки обманул, и эта часть опять про неопределенное поведение...

На самом деле мы получили неопределенное поведение оттого, что случайно нарушили требования `filter`, — мы их уже обсуждали в предыдущей главе.

Как же это всё получилось и кто в этом виноват?

На выходе получилось то, что я ожидал.
Вы получили то, о чем просили.

Эрик Ниблер (создатель `ranges-v3`)

Вернемся к началу. Цепочка `filter | transform` медленнее в 2 раза. Успешные реквесты выполняются два раза. Так и задумано. Из лучших побуждений. И из проблемного дизайна итераторов в C++.

`Range r` в современном C++ — это абстактная штука, у которой есть `begin(r)`, возвращающий итератор `it`, и `end(r)`, возвращающий так называемый `sentinel s` (это еще одна абстрактная штука, с которой сравнивается итератор, чтобы понять, закончилась ли последовательность).

Так что процесс итерации по какому-либо `range` выглядит следующим образом:

```
auto it = begin(r);
auto s = end(r);
while (it != s) {
  do_something_with_element(*it);
  ++it;
}
```

То есть у итератора последовательно вызываются три метода:

- сравнение с `sentinel` — проверка на конец последовательности;
- разыменование — извлечение элемента;
- продвижение к следующему элементу.

Посмотрим, что пишут в стандарте про итератор `views::filter`:

“

26.7.8.3 Class filter_view::iterator [range.filter.iterator]

constexpr range_reference_t<V> operator*() const;
7 Effects: Equivalent to: return *current_;

constexpr iterator & operator++();
9 Effects: Equivalent to:
current_ = ranges::find_if(std::move(++current_),
ranges::end(parent_->base_),
ref(*parent_->pred_));
return *this;

”

Ага, то есть при продвижении итератора выполняется `find_if`, который разыменует `++current_`, чтобы применить к нему предикат фильтрации. И потом мы его опять разыменуем, чтоб достать значение.

Вот оно удвоение!

Но стойте, ведь это ничего не значит. Ну, разыменовываем мы итератор в `filter` два раза. Причем тут повторное исполнение функции `transform`? Может, там ссылка на сохраненый результат возвращается при разыменовании...

Нет, не может. Посмотрим, что пишут в стандарте про `views::transform`.

> 26.7.9.3 Class template transform_view::iterator [range.transform.iterator]
>
> constexpr decltype(auto) operator*() const noexcept(noexcept(invoke(*parent_->fun_, *current_))) {
>
> return invoke(*parent_->fun_, *current_);
>
> }

`transform` всегда вызывает переданную функцию, даже если мы разыменовываем повторно итератор на один и тот же элемент. И по-другому быть не может. Иначе не получится обеспечить никакой `perfect forwarding`, и `transform` не будет работать со всякими странными функциями, возвращающими некопируемые и неперемещаемые объекты. Даже если они мало кому нужны. Стандартная библиотека C++ пытается быть максимально универсальной с совершенно абсурдными последствиями.

Что же делать?

Если вы очень любите стандартную библиотеку, постарайтесь никогда не использовать `views::transfrom` в цепочках перед `filter`, `take_while`, `drop_while` и другими нетривиальными комбинаторами, которым нужен доступ к элементу.

Если вы используете `range-v3` от Эрика Ниблера, как совместимое со стандартной библиотекой решение, используйте `ranges::views::cache1` в конце цепочек `transform`, чтоб не выполнять их повторно.

```
// спасибо @sigasigasiga за предоставленный пример
auto printer = [](auto x) {
    std::println("called");
    return x;
};

auto with_cache = v
    | ranges::views::transform(printer)
```

```
    | ranges::views::cache1
    | ranges::views::filter(is_even)
;
for(auto _ : with_cache);
```

В самой стандартной библиотеке подобный кеширующий `std::views::cache_last` может появиться в C++26 или позже.

Учитывайте, что в таком случае весь `perfect forwarding` заканчивается. В версии Эрика Ниблера возвращается `rvalue`-ссылка, в предложенной `std::`-версии возвращается `lvalue`-ссылка — может потребоваться дополнительное явное перемещение, если нужно избежать копирований.

Также `cache_last`/`cache1` создают дополнительные проблемы, если `transform` должен вернуть ссылку. Вы получите копию:

```
struct Student {
  std::vector<int> grades;
};

auto condition = [](auto&& _) { return true; };

std::vector<Student> v = { Student {  { 1, 2, 3, 4, 5, } } };

std::println("address of grades before: {}\n", uintptr_t(v[0].grades.data()));

auto to_vector = [](std::string&& s) -> std::vector<std::string> {
  return {
    std::move(s)
  };
};

std::println("with cache:");
for (auto&& grades : v
    | ranges::views::transform(&Student::grades)
    // доступ к grades может быть ресурсозатратным
    // (например, если это поиск в большом словаре),
    // поэтому мы предпочитаем кешировать результат
    | ranges::views::cache1
    | ranges::views::filter(condition)) {

    std::println("address of grades after: {}\n", uintptr_t(grades.data()));
}

std::println("without cache:");
for (auto&& grades : v
```

```
    | ranges::views::transform(&Student::grades)
    | ranges::views::filter(condition)) {

    std::println("address of grades after: {}\n", uintptr_t(grades.data()));
}

/* возможный результат
address of grades before: 35328752

with cache:
address of grades after: 35328688 // ! Адрес изменился!

without cache:
address of grades after: 35328752
*/
```

Придется использовать указатели или `std::reference_wrapper`:

```
auto wrap_ref = [](auto& x) { return std::reference_wrapper { x };};
auto unwrap_ref = [](auto ref) -> decltype(auto) { return ref.get(); };

std::println("with cache:");
for (auto&& grades : v
    | ranges::views::transform(&Student::grades)
    | ranges::views::transform(wrap_ref)
    | ranges::views::cache1
    | ranges::views::filter(condition)
    | ranges::views::transform(unwrap_ref)) {
    std::println("address of grades after: {}\n", uintptr_t(grades.data()));
}
/*
// Возможный вывод:
address of grades before: 37319408
with cache:
address of grades after: 37319408 // ! Адрес не изменился, никакого копирования.
*/
```

Если вы не любите стандартную библиотеку, можете реализовать собственную версию, используя дизайн итераторов Rust, Zig, Python (только без исключений) и других языков, где используется ровно один метод и передача по значению:

```
// Rust-вариант страдает от своих особенностей с лайфтаймами,
// которые частично решаются с помощью Generic Associated Types.
// Но как абстрактный пример сойдет
```

```
trait Iterator {
    type Value;
    fn next(&mut self) -> Option<Self::Value>;
}
```

С подобным итератором намного сложнее создать проблему с повторным вызовом методов, но возникают свои трудности, специфичные для C++.

- Что если хочется возвращать ссылки? Ссылки не являются нормальными значениями в C++, и `std::optional<T&>` не компилируется.
- От странных `non-movable`-типов придется отказаться.
- `std::optional` не имеет `niche optimization`.

operator[] для ассоциативных контейнеров

Удивительное дело, но в этом разделе не будет ничего, связанного с неопределенным поведением. По крайней мере напрямую.

В стандартной библиотеке C++ много неоднозначных решений. Одно из таких: для ассоциативного контейнера объединить операцию вставки и получения элемента.

`operator[]` для ассоциативных контейнеров пытается вызвать конструктор по умолчанию для элемента, если не находит переданный ключ.

С одной стороны, это удобно:

```
std::map<Word, int> counts;
for (Word c : text) {
  ++counts[word]; // ровно один поиск по ключу
}
```

В иных языках придется постараться, чтобы записать то же самое и не допустить повторного поиска. В Java:

```
// Поиск трижды!
map.put(key, map.containsKey(key) ? map.get(key) + 1 : 1);

// Поиск дважды!
map.put(key, map.getOrDefault(key, 0) + 1);
```

Оно, конечно, может быть, отоптимизируется JIT-компилятором... Но мы в C++ любим гарантии.

С другой стороны, вызов конструктора, если элемент не найден, может выйти боком:

```
struct S {
  int x;
  explicit S (int x) : x {x} {}
};
```

```
std::map<int, S> m { { 1, S{2} }}; // Ok
m[0] = S(5);   // Огромная трудночитаемая ошибка компиляции.
auto s = m[1]; // Опять огромная трудночитаемая ошибка компиляции.
```

Или другой сценарий:

```
struct Huge {
  Huge() {
    data.reserve(4096);
  }
  std::vector<int> data;
};

std::map<int, Huge> m;
Huge h;
... /* заполняем h */
m[0] = std::move(h); // Бесполезный вызов конструктора по умолчанию,
                     // лишняя аллокация,
                     // а потом перемещение.
```

Чтобы выпутаться из этой неприятности, у ассоциативных контейнеров к C++17 (20) наплодили целую гору методов `insert_or_assign`, `try_emplace` и `insert` с непонятным для непосвященных возвращаемым значением `pair<iterator, bool>`.

Всем этим добром пользоваться тяжело и неудобно. Про них пишут длинные статьи в блоги о том, как эффективно пользоваться поиском по контейнерам...

С `operator[]`, конечно же, проще, "понятнее" и короче. Но это же и ловушка для невнимательных. А если еще и с мерзопакостными особенностями других объектов скрестить...

```
std::map<std::string, std::string> options {
  {"max_value" , "1000"};
}
...
const auto ParseInt = [](std::string s) {
  std::istringstream iss(s);
  int val;
  iss >> val;
  return val;
};

// Перепутали! Нет такого поля!
const int value = ParseInt(options["min_value"]);

// value == 0. Все ОК. Счастливой отладки!
// operator[] вернул пустую строку.
// operator>> ничего не прочел и записал ноль в результат.
```

Избежать неприятностей с operator[] для ассоциативных контейнеров можно, навесив const. И тогда вам этот оператор доступен не будет. И придется использовать либо .at, бросающий исключения, либо всеми любимый:

```
if (auto it = m.find(key); it != m.end()) {
  // Делай, что хочешь, с *it, it->second.
}
```

Всё просто.

std::enable_if_t против std::void_t

Шаблоны C++, начинавшиеся как облагороженная версия копирования/вставки с помощью макроподстановок препроцессора, и обросшие правилами SFINAE, породили довольно жуткие, громоздкие, но мощные возможности для метапрограммирования и вычислений на этапе компиляции.

Механизм невероятно полезный и крайне неудобный. Неудивительно, что в стандартной библиотеке появились различные инструменты для упрощения жизни.

Детали работы SFINAE выходят далеко за пределы этой книги. Здесь же будут обсуждаться новшества в C++17: что должно облегчить написание кода, но не работает.

Если очень кратко, правило SFINAE (substitution failure is not an error) состоит в следующем:

- если при подстановке аргументов в **заголовок** шаблона происходит ошибка (получается невалидная конструкция), то этот шаблон игнорируется, и берется следующий подходящий;
- если больше подходящих шаблонов нет, происходит ошибка компиляции.

Пример:

```
#include <type_traits>

template <class T>
decltype(void(std::declval<typename T::Inner>())) fun(T) { // 1
  std::cout << "f1\n";
}

template <class T>
decltype(void(std::declval<typename T::Outer>())) fun(T) { // 2
  std::cout << "f2\n";
}

struct X {
  struct Inner {};
};
```

```
struct Y {
  struct Outer {};
};

...

fun(X{}); // При подстановке в шаблон 2
          // конструкция X::Outer невалидна:
          // в X нет такого типа. Отбрасывается.
          // Подстановка шаблона 1
          // проходит без ошибок - будет выведено "f1".

fun(Y{}); // Аналогично, но наоборот.
          // Y::Inner не существует. Печатает "f2".
```

Конструкция `decltype(void(std::declval<typename T::Outer>))`, используемая для "паттерн-матчинга", конечно же, ужасна. Сумрачный гений мастеров C++ порождал и более жуткие вещи. Но для менее искушенного пользователя хотелось бы чего-то более простого, понятного и удобного.

Так, у нас есть `std::enable_if_t`, позволяющий триггерить SFINAE не по самописной жуткой конструкции, а по булевому значению.

```
template<class T>
std::enable_if_t<sizeof(T) <= 8> process(T) {
  std::cout << "by value";
}

template<class T>
std::enable_if_t<sizeof(T) > 8> process(const T&) {
  std::cout << "by ref";
}
...
process(5); // по значению
const std::vector<int> v;
process(v); // по ссылке
```

Причем в аргументе `std::enable_if` мы всё так же можем использовать страшные конструкции, а не только какие-то предикаты.

```
template <class T>
std::enable_if_t<std::is_same_v<typename T::Inner, typename T::Inner>>
fun(T) { // 1
  std::cout << "f1\n";
}
```

```
template <class T>
std::enable_if_t<std::is_same_v<typename T::Outer, typename T::Outer>>
fun(T) { // 2
  std::cout << "f2\n";
}

fun(X{}); // Несмотря на то что значение std::is_same_v<T, T>
          // всегда истинно, X::Outer не существует.
          // И SFINAE сработает не из-за значения предиката,
          // а из-за его аргументов.
```

И тут начинается первая неприятность: `std::enable_if` против `std::enable_if_t`.

```
// примерно
template <bool cond, T = void>
struct enable_if {};

template <true, T = void>
struct enable_if {
  using type = T;
};

template <bool cond, T = void>
using enable_if_t = typename enable_if<cond, T>::type;
```

Они легко путаются при быстром наборе с автокомплитом. Стоит случайно опустить суффикс `_t`, и вместе с ним будут потеряны многие часы отладки всего этого сопоставляющего с образцами добра:

```
// SFINAE триггерилось от значения предиката и больше не работает.
// std::enable_if<false> - валидный тип.
// Получаем UB из-за переопределения одной и той же сущности по-разному.
template<class T>
std::enable_if<sizeof(T) <= 8> process(T);
template<class T>
std::enable_if<sizeof(T) > 8> process(const T&);

// SFINAE триггерилось от аргументов предиката и
// продолжает работать.
// Если ожидали void в качестве типа возврата,
// может быть UB из-за отсутствующего return.
template <class T>
```

```
std::enable_if<std::is_same_v<
  typename T::Inner, typename T::Inner>> fun(T);
template <class T>
std::enable_if<std::is_same_v<
  typename T::Outer, typename T::Outer>> fun(T);
```

Эта же неприятность касается всех остальных зверей из заголовка <type_traits>. Любые std::trait_X и std::trait_X_t являются типами, которые, будучи перепутанными, далеко не всегда проявляют себя.

Лучше взять за правило: с помощью std::enable_if триггерить SFINAE только по предикату. Так проблем будет меньше.

Если предиката нет, его можно написать:

```
template <class T,
          class = void> // костыль-заглушка для проверяемого "паттерна"
struct has_inner_impl : std::false_type {};

template <class T>
struct has_inner_impl<T,
   // сам "паттерн", тип-результат должен совпадать с тем,
   // что указан в заглушке выше
   decltype(void(std::declval<typename T::Inner>()))>
   : std::true_type {};

template <class T>
constexpr bool has_inner_v = has_inner_impl<T>::value;

static_assert(has_inner_v<X>);
static_assert(!has_inner_v<Y>);
```

Это один из наиболее распространенных и "простых" подходов к написанию подобных предикатов. А void чаще всего используется в качестве костыльной заглушки. И чтобы не писать этот страшный decltype(void(std::declval<X>())) каждый раз, когда нам нужно всего лишь проверить тип X на валидность, придумали, а потом и втащили в C++17 шаблон std::void_t.

Вот такой:

```
template <class...>
using void_t = void;
```

И с ним всё должно стать короче и красивее:

```
template <class T>
struct has_inner_impl<T,
  std::void_t<typename T::Inner>>
  : std::true_type {};
```

Но, увы, это чаще не работает, чем работает. В стандарте C++11 был обнаружен дефект (issue 1558 — WG21 CWG Issues), позволяющий этой конструкции не работать. И под многими не самыми-самыми новыми версиями компиляторов такой предикат всегда будет возвращать истину.

Но и под новыми версиями `std::void_t` также сломан. Попробуем использовать его, чтобы переписать самый первый пример в этом разделе:

```
template <class T>
std::void_t<typename T::Inner> fun(T) {
  std::cout << "f1\n";
}

template <class T>
std::void_t<typename T::Outer> fun(T) {
  std::cout << "f2\n";
}
```

Ни один из трех основных компиляторов (GCC, Clang, MSVC) не будет этот код компилировать, несмотря на то что первая версия с уродливым `decltype` собиралась.

Потому что у нас есть понятия "эквивалентности" и "функциональной эквивалентности" объявлений, компилятор проверяет первое. А вот второе имеет отношение к SFINAE. Страшная вещь.

Цитата из "C++ Standard Core Language Active Issues, Revision 114":

> 1980. Equivalent but not functionally-equivalent redeclarations. In an example like:
>
> template<typename T, typename U> using X = T;
>
> template<typename T> X<void, typename T::type> f();
>
> template<typename T> X<void, typename T::other> f();
>
> it appears that the second declaration of f is a redeclaration of the first but distinguishable by SFINAE, i.e., equivalent but not functionally equivalent.
>
> Notes from the November, 2014 meeting: CWG felt that these two declarations should not be equivalent.

Общий совет: не пользоваться `std::void_t`, а также не пытаться строить SFINAE на параметрах шаблонов-алиасов, если от этих параметров ничего не зависит справа от `using =`.

```
template <class T>
struct my_void {
  using type = void;
}
```

```
template <class T>
using my_void_t = void; // не работает

template <class T>
using my_void_t = typename my_void<T>::type; // OK
```

А вообще, лучше переходить на C++20 и не заниматься всей этой ерундой. Там специально для всех этих страшных конструкций читаемый синтаксический сахар придумали. Конечно, не без затаившихся граблей, но об этом в другой раз.

Тип std::aligned_storage

Всех C++ разработчиков можно довольно успешно разделить на две категории:

- тех, у кого код вида

  ```
  char buff[sizeof(T)];
  ...
  T* obj = new (buff) T(...);
  ```

  работает, и для них нет никаких проблем;
- и тех, у кого из-за такого кода внезапно прилетает SIGSEGV или SIGBUS, или что-то еще более интересное.

Чаще всего первые собирают свои программы только под *x*86 каким-нибудь старым компилятором, ничего не знающим про SIMD-инструкции, и, вероятно, с выключенными "агрессивными" оптимизациями, чтобы точно ничего не сломалось.

Разработчики на C тоже не должны уйти обиженными. Для них этот код будет выглядеть как:

```
char buff[sizeof(T)];
...
T* obj = (T*)buff;
```

что, в принципе, еще страшнее из-за неинициализированной памяти, но это уже другая история.

Основная проблема: буфер, в который мы тут собрались что-то записать, может быть выровнен (*alignment*) не так, как это требуется для типа `T`.

О том, что размер укладываемых в буфер данных должен быть не больше размера самого буфера, многие узнают довольно быстро и также быстро вспоминают, хотя бы из соображений здравого смысла. А вот с выравниванием всё сложно.

Чтобы бездушная машина могла успешно прочитать/записать данные типа `T` по адресу, соответствующему значению указателя `T* ptr`, или иногда совершить какую-то хитрую операцию над ними, адрес должен быть выровнен — кратен некоторому числу (обычно это степень двойки), которое нам навязали инженеры-разработчики микроархитектуры этой самой машины.

А навязали, потому что:

- так должно быть эффективнее;
- по-другому не получалось;
- надо было очень существенно сэкономить на длине инструкции (чем больше выравнивание, тем больше младших битов адреса можно не использовать);
- добавьте свою причину, если вы проектируете набор инструкций и знаете, что еще можно придумать.

Если вернуться к C++, то обычно мы знаем выравнивание, требуемое для встроенных типов:

Тип	Выравнивание
char	1
int16	2
int32	4
int64	8
__m128i	16

Хотя как знаем... Например, тип double вполне себе стандартный и имеет размер 8 байт — что на 32-битных Linux-системах, что на 64-битных. Но неожиданно выравнивание может оказаться разным. На 32-битной системе double выравнивается по границе 4 байта, а в 64-битной системе — по границе 8 байт. Так и живем.

Для других типов, специализированных для работы с SSE/SIMD-инструкциями, выравнивание может быть и больше.

Для пользовательских структур и классов наследуется наибольшее выравнивание из всех полей. А между полями появляются неявные байты-заполнители (*padding*), чтобы удовлетворять требованиям выравнивания каждого поля по отдельности.

```
struct S {
  char c;    // 1
  // Неявно char _padding[3]
  int32_t i; // 4
};

static_assert(alignof(S) == 4);
```

Выравнивание массива соответствует выравниванию его элементов.

Поэтому:

```
char buff[sizeof(T)]; // alignment == 1
...
T* obj = new (buff) T(...); // (uintptr_t)(obj) должен быть
// Кратен alignof(T), но в этом коде гарантируется
// только то, что он кратен 1.
```

Для встроенных типов на *x*86 доступ по невыровненным указателям чаще всего просто приводит к более медленному исполнению. На других же платформах может быть segfault. Искать такие проблемы можно, например, с помощью анализатора PVS-Studio: V1032 Pointer is cast to a more strictly aligned pointer type.

С SSE-типами и на *x*86 можно легко получить segfault, причем довольно красиво:

```
#include <memory>
#include <xmmintrin.h>

const size_t head = 0;
struct StaticStorage {
  char buffer[256];
} storage;

int main() {
  __m128i* a = new (storage.buffer) __m128i();
  // Закомментируйте строку выше и раскомментируйте любую
  // из строк ниже, чтобы получить segfault с clang-16 -O0
  // __m128i* b = new (storage.buffer + 0) __m128i();
  // __m128i* c = new (storage.buffer + head) __m128i();
}
```

Что же делать?! Как же написать код без такого интересного неопределенного поведения? Не волнуйтесь, C++11 спешит на помощь!

Стандарт предоставляет alignas-спецификатор. С его помощью можно явно указать требования к выравниванию при описании переменных и структур.

```
#include <memory>
#include <xmmintrin.h>

const size_t head = 0;
struct StaticStorage {
  alignas(__m128i) char buffer[256];
} storage;

int main() {
  __m128i* a = new (storage.buffer) __m128i();
  __m128i* b = new (storage.buffer + 0) __m128i();
  __m128i* c = new (storage.buffer + head) __m128i();
}
```

И вот уже программа и не падает, но выглядит как-то громоздко. Неужели для такого важного случая, как создание буфера с подходящим размером и выравниванием, в стандартной библиотеке C++ нет никакой удобной функции?

Конечно, есть!

`std::aligned_storage` и его старший брат `std::aligned_union`:

```
template<std::size_t Len,
         std::size_t Align =
           /* default alignment not implemented */
        >
struct aligned_storage
{
  struct type
  {
    alignas(Align) unsigned char data[Len];
  };
};

template <std::size_t Len, class... Types>
struct aligned_union
{
  static constexpr std::size_t alignment_value =
                                 std::max({alignof(Types)...});

  struct type
  {
    alignas(alignment_value)
      char _s[std::max({Len, sizeof(Types)...})];
  };
};
```

Это практически то же самое, что было в примере выше! Первый — совсем низкоуровневый, ему нужно напрямую число-значение выравнивания указать. А второй — более умный, он сам подходящее значение по списку типов выберет. Да еще и размер буфера подстроит, если мы неправильный указали. Какая удобная метафункция!

Давайте же ею воспользуемся:

```
#include <memory>
#include <xmmintrin.h>
#include <type_traits>

const size_t head = 0;
std::aligned_union<256, __m128i> storage;

int main() {
  __m128i* a = new (&storage) __m128i();
```

```
  __m128i* b = new ((char*)(&storage) + 0) __m128i();
  __m128i* c = new ((char*)&storage + head) __m128i();
}
```

И сразу же всё упало: SIGSEGV.

Но как же так?! Ведь всё же верно...

Давайте-ка проверим.

```
static_assert(sizeof(storage) >= 256);
<source>:9:1: error:
static assertion failed due to requirement
'sizeof (storage) >= 256'
static_assert(sizeof(storage) >= 256);
^             ~~~~~~~~~~~~~~~~~~~~~
<source>:9:31: note: expression evaluates to '1 >= 256'
static_assert(sizeof(storage) >= 256);
```

Дивно! Но если мы еще раз внимательно посмотрим на примеры определения шаблонов `std::aligned_storage` выше, то обнаружим великую подлость и предрасположенность этих шаблонов к ошибке использования.

Нам нужно использовать `std::aligned_union<256, __m128i>::type storage`!

Или `std::aligned_union_t<256, __m128i> storage` в C++17.

Теперь всё работает. Разница всего в два символа, а какие последствия.

На момент написания книги GCC 14.1 способен из коробки выдать предупреждения:

```
<source>:12:23: warning:
placement new constructing an object of type '__m128i' and
size '16' in a region of type
'std::aligned_union<256, __vector(2) long long int>' and
size '1' [-Wplacement-new=]
   12 |     __m128i* a = new (&storage) __m128i();
```

наводящие на мысль об ошибке.

Компилятор Clang 18.1 по умолчанию такого не сообщает.

`std::aligned_*` признаны невероятно опасными к использованию. Из-за ужасного дизайна, ошибки при его использовании очень легко прячутся.

В C++23 их пометили как устаревший (deprecated). Но кто ж знает, когда мы увидим C++23 в больших и старых кодовых базах...

Если вы используете `std::aligned_*` в своем коде, то убедитесь дважды, что применяете его правильно. А лучше замените своей структурой с явным использованием `alignas`.

Перегруженные конструкторы стандартной библиотеки

При проектировании стандартной библиотеки C++ было принято множество странных решений, из-за которых приходится страдать. И исправить их не представляется возможным из-за соображений обратной совместимости.

Одним из таких странных решений являются перегрузки конструкторов с радикально различным поведением.

Яркий пример:

```
using namespace std::string_literals;
std::string s1 { "Modern C++",  3 };
std::string s2 { "Modern C++"s, 3 };

std::cout << "S1: " << s1 << "\n";
std::cout << "S2: " << s2 << "\n";
```

Этот код выведет:

```
S1: Mod
S2: ern C++
```

Потому что у `std::basic_string` есть один конструктор, принимающий указатель и длину строки. А есть еще один конструктор, принимающий "что-то похожее на строку" и позицию, с которой надо из нее извлечь подстроку!

На этом причуды не заканчиваются.

```
std::string s1 {'H', 3};
std::string s2 {3, 'H'};
std::string s3 (3, 'H');

std::cout << "S1: " << s1.size() << "\n";
std::cout << "S2: " << s2.size() << "\n";
std::cout << "S3: " << s3.size() << "\n";
```

Этот пример выведет:

```
S1: 2
S2: 2
S3: 3
```

Потому что у строки есть конструктор, принимающий число `n` и символ `c`, который нужно повторить `n` раз. А еще есть конструктор, принимающий список инициализации (`std::initializer_list<T>`), состоящий из символов. И существование этого конструктора взаимодействует с неявным приведением типов!

- `std::string s1 {'H', 3};` — строка `"H\3"`.
- `std::string s2 {3, 'H'};` — строка `"\3H"`.
- `std::string s3 (3, 'H');` — строка `"HHH"`.

Аналогичной проблемой страдает std::vector:

```
std::vector<int> v1 {3, 2}; // v1 == {3, 2}
std::vector<int> v2 (3, 2); // v2 == {2,2,2}
```

А еще у контейнеров есть конструктор, принимающий пару итераторов. И, казалось бы, с ними уж проблем-то не будет, но у нас есть указатели, которые также являются итераторами. А еще есть тип bool:

```
bool array[5] = {true, false, true, false, true};
std::vector<bool> vector {array, array + 5};
std::cout << vector.size() << "\n";
```

Будет выведено 2, а не 5, потому что указатели неявно приводятся к bool!

Благо, с приходом C++20 преобразование указателей в bool стали считать сужающим, причем даже и в предыдущих редакциях стандарта. Так что последние версии компиляторов начали либо, как GCC, по умолчанию выдавать предупреждение:

```
narrowing conversion of '(bool*)(& array)' from 'bool*' to 'bool'.
```

либо, как Clang, отказывают в компиляции:

```
error: type 'bool *' cannot be narrowed to
'bool' in initializer list [-Wc++11-narrowing]
```

Собственно, эти прекрасные примеры показывают, почему "универсальная" инициализация не универсальная.

Чтобы не множить хаос в своих проектах, нужно быть осторожнее с объявлением перегруженных конструкторов для своих типов. Лучше ввести статическую функцию, чем создавать перегруженные конструкторы, неожиданно взаимодействующие с неявным приведением типов и списками инициализации.

Класс std::uniform_int_distribution

> Нельзя так просто взять и сгенерировать случайную последовательность байтов.

В C++11 стандартная библиотека пополнилась многими отличными штуками, в том числе функциями и классами для работы с генераторами случайных чисел удобным и, как это ни странно, предсказуемым способом.

C++ в наследство от C досталась функция rand(), которой на сегодняшний день не рекомендуется пользоваться нигде, кроме как в совершенно игрушечных проектах.

- Если ее seed не инициализировать через srand(), он будет инициализирован, внезапно, единицей (а могли бы взять 42).
- Естественно, функция rand() использует некое глобальное, возможно, "thread local"-состояние. Это implementation defined.
- В многопоточной среде безопасность потоков (thread safety) — тоже implementation defined. В общем, очень непредсказуемая штука!

Другое дело функционал из #include <random>!

Вот вам и разные генераторы, и преобразования функций распределения, и вы их можете друг от друга отделять и иметь собственный, особенно инициализированный генератор для каждой отдельной сущности в вашем проекте, и многопоточный доступ организовывайте, как хотите. Красота!

Давайте-ка сгенерируем последовательность случайных, равномерно распределенных байтов.

```
#include <iostream>
#include <random>

int main()
{
  std::random_device rd;  // Источник для посева значений в генератор случайных чисел
  std::mt19937 gen(rd()); // mersenne_twister_engine засеян rd()
  std::uniform_int_distribution<uint8_t> distrib{}; // Не пытайтесь использовать
                                                    // std::byte! Не скомпилируется.

  // Сгенерируйте случайный байт при помощи distrib
  for (int n = 0; n != 10; ++n)
    std::cout << int32_t(distrib(gen)) << ' '; // Нужно выполнить приведение,
                                               // чтобы выводить числовые значения,
                                               // а не символьные.
  std::cout << '\n';
}
```

Компилируем, запускаем, всё отлично работает!.. По крайней мере с GCC и Clang. А что если нам нужна кроссплатформенная сборка, в том числе под Windows, с помощью MSVC?.. Давайте попробуем просто взять и скомпилировать как есть...

Ой...

```
example.cpp
C:/data/msvc/14.39.33321-Pre/include\random(2107): error C2338: static_assert failed:
'invalid template argument for uniform_int_distribution: N4950 [rand.req.genl]/1.5
requires one of short, int, long, long long, unsigned short, unsigned int, unsigned
long, or unsigned long long'
C:/data/msvc/14.39.33321-Pre/include\random(2107): note: the template instantiation
context (the oldest one first) is
<source>(9): note: see reference to class template instantiation
'std::uniform_int_distribution<uint8_t>' being compiled
C:/data/msvc/14.39.33321-Pre/include\random(2107): error C2338: static_assert failed:
'note: char, signed char, unsigned char, char8_t, int8_t, and uint8_t are not allowed'
Compiler returned: 2
```

Ну-у, матерый разработчик, знакомый с причудами старых версий MSVC, который с включенным по умолчанию \permissive+ мог компилировать совершенно безум-

ные вещи[10], не сильно удивится и скажет, что опять в Microsoft какую-то ерунду придумали...

Удивительно, но нет!

```
Throughout this subclause [rand], the effect of instantiating a template:
...
that has a template type parameter named UIntType is undefined unless the
corresponding template argument is cv-unqualified and is one of unsigned short,
unsigned int, unsigned long, or unsigned long long.
```

Подставлять `uint8_t` в `std::uniform_int_distribution` стандартом не разрешается под страхом неопределенных эффектов.

Это, конечно, совершенно нелепо, и в 2013 году даже поднимался вопрос о принятии defect report, но его отклонили как not a defect и предложили написать proposal в стандарт, чтобы как-то это дело исправить... В общем по состоянию на 2025 год не исправили. Возможно, исправят в C++26. Или в C++29[11].

Кстати, на всякий случай, `__int128_t` тоже использовать как бы нельзя. Хотя с GCC работает.

Функция std::vector::reserve и метод std::vector::resize

Очень многие программы на C++ становятся жертвами подлого обмана со стороны этих двух братьев-близнецов. А также, естественно, невнимательности, спешки и автодополнения.

Почти все книги по программированию на C++ учат, что при создании `std::vector` желательно заранее резервировать память, особенно если вы знаете, сколько у вас будет элементов в нем. Тогда при наполнении вектора он не будет реаллоцироваться, а значит, ваша программа заработает быстрее, не тратя время на перевыделение памяти.

Но вот беда — у `std::vector` нет конструктора, в котором можно было бы указать: "Хочу пустой вектор, но с зарезервированной capacity = *N*". У вектора есть другой конструктор — заполняющий *N* элементами по умолчанию:

```
// Создаст N пустых строк.
std::vector<std::string> text(N);
```

Но это ведь неоптимально. Создать целый вектор, проинициализировать каждый элемент в нем, чтоб потом переписать их... Нет-нет, это неправильное использование C++!

У нас есть метод `reserve()`, его надо вызвать после создания пустого вектора.

```
std::vector<std::string> text;
text.reserve(N);
```

[10] Например, можно передать временное значение в функцию, принимающую неконстантную ссылку!
[11] proposal в 2024 году пока еще не был написан! Это ваш шанс!

Но снова напасть! Ведь у нас есть еще и метод `resize()`, который также может принять ровно один аргумент, если элементы вектора конструируемы по умолчанию.

И, разумеется, программисты их путают:

- имена короткие и начинаются одинаково;
- имеют схожий смысл;
- стоят рядом в выдаче автодополнения;
- и в наши дни еще и ИИ-ассистент[12] может не того из них посоветовать!

В итоге программист успешно создаст вектор в два раза больше, чем хотелось, и будет долго недоумевать: почему у него все интересующие элементы пустые?

```
auto read_text(size_t N_lines) {
  std::vector<std::string> text;
  text.resize(N_lines);    // Ай!
  for (size_t line_no = 0; line_no < N_lines; ++line_no) {
    std::string line;
    std::getline(std::cin, line);
    text.emplace_back(std::move(line));
  }
  return text;
}
```

Но никакого неопределенного поведения. Эх!

Но что если программист написал `reserve()` там, где на самом деле требовался `resize()`? Ну, случайно. Причем эта случайность имеет довольно неплохой шанс: ведь выдача автодополнения часто упорядочена по алфавиту, а `reserve` стоит в нем раньше.

```
std::pair<std::vector<std::byte>, size_t>
  read_text(std::istream& in, size_t buffer_len)
{
  std::vector<std::byte> buffer;
  buffer.reserve(buffer_len);
  in.read(reinterpret_cast<char*>(buffer.data()), buffer_len);
  return {
    std::move(buffer), static_cast<size_t>(in.gcount())
  };
}

int main() {
  auto [buffer, actual_size] = read_text(std::cin, 45);
```

[12] ИИ — искусственный интеллект.

```
  for (size_t i = 0; i < actual_size; ++i) {
    std::cout << static_cast<int>(buffer[i]) << "\n";
  }
}
```

Программа будет успешно работать!

```
// Пример ввода/вывода
>> hello
104
101
108
108
111
```

И вот здесь сто́ит на минуту притормозить и отвлечься. Я множество раз видел на самых разных технических форумах заявления вида:

- C и C++ — это языки для работы близко к железу;
- undefined behavior — это просто формальность;
- если программист знает, как работает память, как работает его программа и типы, он может эту ерунду игнорировать,

и так далее.

Случай с `reserve()` выше можно использовать в защиту такой спорной позиции. Действительно, я знаю, что:

- `reserve()` на самом деле выделяет память, так что полуинтервал `[buffer.data(), buffer.data() + buffer_len)` валиден;
- `std::istream::read` проинициализировал память в полуинтервале `[buffer.data(), buffer.data() + actual_size);`
- `operator[]` вектора по умолчанию не проверяет переданный индекс;
- итерации цикла доступа к вектору проходят в инициализированных пределах.

Поэтому всё работает. Более того, оно даже работает без language undefined behaviour. Если вы скопируете реализацию `std::vector` к себе, удалите из нее все `asserts` с условными инструментациями санитайзеров и станете пользоваться вот таким же странным образом, у вас в коде не будет неопределенного поведения. По крайней мере в этом примере.

Но! Это `std::vector. std`! И он декларирует library undefined behaviour — вы обратились к элементу с индексом пределами `[0, vector::size())`. А это не прощается.

Я не смог найти ни одного компилятора, доступного онлайн, на котором можно было бы воспроизвести последовательность оптимизаций, приводящих к падению невероятной красоты. Но я видел несколько закрытых bug-репортов в отношении Apple Clang, который такое проворачивал.

LLVM может генерировать под *x*86 инструкцию `ud2` — это недопустимая инструкция, часто используемая как индикатор недостижимого кода. Если программа по-

пытается ее выполнить, она умрет от сигнала SIGILL. Код, который провоцирует неопределенное поведение, может быть помечен как недостижимый и в дальнейшем заменен на ud2 или выброшен. В нашем замечательном примере компилятору вполне известно, что `buffer.size() == 0`. И его не меняли.

Так, например, если мы попробуем один в один переписать это же безобразие в Rust, агрессивно использующем возможности LLVM:

```
fn read(n: usize, mut reader: impl std::io::Read) ->
  std::io::Result<(Vec<u8>, usize)>
{
  // Резервируем память. Она будет неинициализированной.
  let mut buf = Vec::<u8>::with_capacity(n);

  // unsafe Rust довольно сложен, и формально здесь
  // нельзя напрямую создавать &mut [u8]
  // на неинициализированную память.
  // Но "мы знаем что делаем" - на результат это не повлияет.
  // Пока...
  let actual_size = reader.read(unsafe {
      std::slice::from_raw_parts_mut(buf.as_mut_ptr(), n)
  })?;
  // В отличие от C++, в Rust можно сделать
  // unsafe { buf.set_len(actual_size) };
  // И сделать этот пример практически корректным.
  // Но мы здесь собрались смотреть на undefined behavior.
  Ok((buf, actual_size))
}

pub fn main() {
  let (buf, n) = read(42, std::io::stdin()).unwrap();
  for i in 0..n {
    println!("{}",
      unsafe { buf.get_unchecked(i) }
    )
  }
}
```

В дебажной сборке мы упадем с ошибкой сегментации и сообщением

```
thread 'main' panicked at library/core/src/panicking.rs:220:5:
unsafe precondition(s) violated: slice::get_unchecked requires
that the index is within the slice
note: run with `RUST_BACKTRACE=1` environment variable to
```

```
display a backtrace thread caused non-unwinding panic. aborting.
Program terminated with signal: SIGSEGV
```

В релизной с `-C opt-level=3` мы увидим пустой вывод и успешный выход. И если посмотрим на сгенерированный код, то цикла мы в нем не обнаружим. Код обращения к элементам вектора был полностью выброшен как недостижимый. Спасибо аннотации `assert_unsafe_precondition!(check_language_ub, ...)`.

```
example::main::h67df0b7f9b5f8d1a:
      ...
      call    qword ptr [rip + <std::io::stdio::Stdin as
                std::io::Read>::read::h30ce8d6974df759c@GOTPCREL]
      mov     esi, 42
      test    rax, rax
      jne     .LBB1_3    # Это unwrap
      mov     edx, 1
      mov     rdi, rbx
      call    qword ptr [rip + __rust_dealloc@GOTPCREL]
      add     rsp, 8
      pop     rbx
      pop     r14
      ret
.LBB1_7:
```

"Ну это же Rust!" — можно возразить, закатывая глаза. Да. Но это лишь дело времени, когда Clang начнет применять те же оптимизации к C++.

Что же делать?!

По-хорошему, конечно, не ошибаться и не путать `reserve()` и `resize()`...

Как выяснилось после множества экспериментов с разными утилитами, состояние диагностики подобного standard library level неопределенного поведения в C++ в 2025 году остается весьма плачевным:

- статические анализаторы, к сожалению, молчат;
- санитайзеры по умолчанию тоже не реагируют;
- `_ITERATOR_DEBUG_LEVEL` от MSVC молчаливо падает;
- `-fsanitize=address` перестает молчать только лишь с `-stdlib=libc++ ==1==ERROR: AddressSanitizer: container-overflow on address 0x504000000050 at pc 0x59481461d1b0 bp 0x7ffcf01b08b0 sp 0x7ffcf01b08a8 READ of size 1 at 0x504000000050 thread T0`.

Стойте-стойте! А что если это не ошибка. Мы сознательно использовали `reserve()`, т. к. он не инициализирует память, хотели ее, как в примере с Rust, переписать какими-нибудь данными из файла и в конце изменить `size()`. Но вектор просто не предоставляет такое API...

На этот случай в стандартной библиотеке C++ есть аж два более корректных способа.

Применение std::make_unique_for_overwrite

```
std::pair<std::unique_ptr<std::byte[]>, size_t>
read_text(std::istream& in, size_t buffer_len)
{
  // Выделяем инициализированный по умолчанию буфер,
  // но инициализация массива тривиальных объектов по умолчанию - это
  // отсутствие инициализации.
  auto buffer = std::make_unique_for_overwrite<std::byte[]>(buffer_len);
  in.read(reinterpret_cast<char*>(buffer.get()), buffer_len);
  return {
    std::move(buffer), static_cast<size_t>(in.gcount())
  };
}
```

Такой вариант так же успешно работает (**https://godbolt.org/z/qrh9xferr**), как и первоначальный неправильный, но уже без неопределенного поведения.

Но разумеется, мы таким образом запросто потеряли информацию об оставшейся вместимости. Ведь она не привязана к `unique_ptr`. Нужно привязать ее отдельно в рамках собственной структуры. Или забыть.

Функция-член std::basic_string::resize_and_overwrite

Да! Чудо случилось, и в C++23 мы можем сделать почти так же замечательно эффективно, как в Rust. Но только для "строк". Но ведь по старой доброй традиции из C у нас строки — это просто последовательность байтов...

```
// Придется добавить немного CharTraits магии,
// если мы хотим использовать
// std::basic_string с типом std::byte.
struct ByteTraits {
  using char_type = ::std::byte;
  static char_type* copy(char_type* dst,
                         char_type* src, size_t n) {
    memcpy(dst, src, n);
    return dst;
  }
  static void assign(char_type& dst, const char_type& src) {
    dst = src;
  }
};

std::basic_string<std::byte, ByteTraits>
  read_text(std::istream& in, size_t buffer_len)
```

```
{
  std::basic_string<std::byte, ByteTraits> buffer;
  buffer.resize_and_overwrite(buffer_len,
                              [&in](std::byte* buf, size_t len) {
    in.read(reinterpret_cast<char*>(buf), len);
    return static_cast<size_t>(in.gcount());
  });

  return buffer;
}

int main() {
  auto buffer = read_text(std::cin, 45);
  size_t actual_size = buffer.size();
  std::cout << actual_size << std::endl;
  for (size_t i = 0; i < actual_size; ++i) {
    std::cout << static_cast<int>(buffer[i]) << "\n";
  }
}
```

О чудо! Оно также работает, как ожидается.

Класс-шаблон std::function

C++11 предоставил разработчикам очень удобный класс-шаблон для описания абстрактных вызываемых объектов:

```
std::function<R(Args)> f = /* Всё, что угодно,
                              что можно вызвать как f(args),
                              и результатом будет R. */
```

Благодаря технике type-erasure (стирание типа) `std::function` может хранить в себе всё, что угодно. У этого, конечно, есть цена — посредственная производительность: конкретный вызываемый объект должен быть перемещен в кучу, выделение памяти, динамическая диспетчеризация вызова... Если мы не пишем чего-то высоконагруженного, то цена не очень высока.

Однако благодаря тому же самому стиранию типов и тому, как оно реализовано, `std::function` обладает еще некоторыми потрясающими спецэффектами!

Спецэффект 1: вариантность

С этим понятием знакомы далеко не все разработчики, так что начнем с примера.

Пожалуй, я не позволю себе использовать набивший оскомину пример с классами `Animal` и `Dog`. Вместо этого у меня будут `InputDevice` и `Keyboard`. Вопрос: "Если

Keyboard — это подтип InputDevice, является ли Containter<Keyboard> подтипом Container<InputDevice>?"

С точки зрения абстрактной достаточно высокоуровневой иерархии в системе типов — вполне себе. Коробка клавиатур — это коробка устройств ввода. И так, например, будет в языках Java, Kotlin или Rust:

```
trait InputDevice {
    fn input(&self) -> char;
}

trait Keyboard: InputDevice {
    fn lock_keys(&self);
}

struct MBox<T>(T);

// Коробка клавиатур - это коробка устройств ввода!
// Keyboard          - это подтип InputDevice
// MBox<Keyboard>    - это подтип MBox<InputDevice>
fn relabel(b: MBox<impl Keyboard>) -> MBox<impl InputDevice> {
    b
}
```

Теоретики скажут, что коробка **ко**вариантна типу содержимого! Типы контейнеров и типы содержимого вложены согласованно. Поэтому ковариантна.

А бывает и обратная ситуация. Например, совсем не факт, что работник, который умеет чинить клавиатуры, способен чинить произвольные устройства ввода. А вот если наоборот — то без проблем. Он и клавиатуру вам починит.

```
trait Senior {
    fn repair(&self, _: impl InputDevice);
}
trait Junior {
    fn repair(&self, _: impl Keyboard);
}
impl <M: Senior> Junior for M {
    fn repair(&self, k: impl Keyboard) {
        Senior::repair(self, k)
    }
}

// Keyboard                          - это подтип InputDevice
// Senior (ремонтирует InputDevice) - это подтип Junior
```

```
//                                        (который ремонтирует только клавиатуры)
fn demote(m: impl Senior) -> impl Junior {
    m
}
```

Вложенность типов работников противоположна вложенности типов объектов, с которыми они работают. Они контравариантны.

Шаблоны в C++ инвариантны: отношения между типами-контейнерами никак не зависят от типов элементов... Так по крайней мере должно быть, и так и задумывалось. Если бы не внезапное исключение в виде `std::function`!

```
class InputDevice {
public:
    virtual ~InputDevice() = default;
    virtual char input() = 0;
};

class Keyboard : public InputDevice {
public:
    char input() override {
        return '*';
    }
};
// Мы можем сделать так:
Keyboard* keyboard = ...;
// ведь клавиатура - это устройство ввода.
InputDevice* device = keyboard;

std::vector<Keyboard*> box_of_keyboards { keyboard, keyboard, keyboard };
// Ошибка компиляции. Шаблоны инвариантны. Такого оператора присваивания нет.
std::vector<InputDevice*> box_of_input_devs = box_of_keyboard;

// Но!
std::function<Keyboard* ()> keyboard_maker = ...;
// Отлично компилируется! std::function ковариантен по возвращаемому значению.
std::function<InputDevice* ()> input_device_maker = keyboard_maker;
std::function<void(InputDevice*)> senior = ...;
// Тоже компилируется! std::function контравариантен по принимаемым параметрам.
std::function<void(Keyboard*)> junion = senior;
```

С одной стороны, выглядит очень даже здорово и правильно. А с другой стороны, оно, разумеется, работает не потому, что std::function на самом деле поддерживает вариантность. Так происходит из-за неявного приведения типов аргументов и возвращаемого значения, а также повторного стирания типов (со всеми накладными расходами).

```
std::function<void(InputDevice*)> senior = [](auto){}; /* Аллокация и перемещение лямбды на кучу! Стерли тип лямбды внутри */
std::function<void(Keyboard*)> junion = std::move(senior); /* Типы разные. Шаблоны инвариантны. Еще одна аллокация! и перемещение исходной std::function на кучу. Стираем ее тип. */
```

А если цепочки передачи таких функций с изменением типов будут более длинными, становится уже не так здорово.

Проблему можно усугубить тем, что такая "вариантность" работает не только с указателями/ссылками на наследуемые классы в сигнатуре функции. Как было замечено ранее, оно работает из-за неявного приведения типов. А значит, мы можем сделать так:

```
std::function<void(std::string_view)> by_view = [](std::string_view v){
    std::cout << v;
};
std::function<void(const std::string&)> by_str_ref = by_view;
std::function<void(std::string)> by_str_val = by_str_ref;
std::function<void(const char*)> by_char_ptr = by_str_val;
by_char_ptr("hello");
```

И если туда попадет nullptr... Мы помним что будет[13]:

```
Program returned: 139
Program stderr
terminate called after throwing an instance of 'std::logic_error'
  what():  basic_string: construction from null is not valid
Program terminated with signal: SIGSEGV
```

И разумеется, при наличии таких цепочек сохранять переданные в аргументах ссылки становится особенно сомнительным занятием.

Проблему с переизбытком аллокаций C++26 предлагает решать с помощью std::function_ref — невладеющих ссылок на вызываемый объект. Создайте ваш объект один раз и храните, а дальше передавайте на него ссылку с удобным интерфейсом. Вариантность остается в комплекте с возможностью получить dangling reference на другой std::function_ref в цепочке присваиваний.

[13] Конструирование std::string от nullptr не определено. Но основные реализации бросают исключение.

Спецэффект 2: отломанный const

Усердно стирая типы, `std::function` перестарался и подтер пробрасывание `const`.

```
int main() {
  const auto counter = [count = 0]() mutable {
    return count++;
  };
  // Вызов невозможен: для мутабельной лямбды требуется доступ
  // с правами на изменение operator()
  // counter();
  const std::function<int()> f = counter;
  f(); // Это нормально! const не просачивается!
}
```

Ну подтер и подтер... Ничего ж страшного. Код компилируется и вызывается — это главное! А некорректный код, который тоже компилируется из-за этого, просто не нужно писать...

О важности правильного пробрасывания `const`, а особенно с приходом C++23 и deducing this, можно вспомнить один забавный факт о популярном фреймворке Qt: в нем собственные контейнеры с Copy-on-Write-поведением по умолчанию. И это приводит к тому, что совершенно очевидная read-only-итерация по контейнеру внезапно вызывает копирование всего контейнера! А также к инвалидациям ссылок и другим занятным последствиям и порче памяти.

```
// Бах! range based for вызывает begin()/end() методы,
// которые в не-const-версии вызывают копирование!
for (const auto& x: qlist) {};
// Вот так надежнее и корректнее:
for (const auto& x: std::as_const(qlist)) {};
```

Если мы теперь вернемся к `std::function` и будем передавать в нее вызываемый объект с существенно разными `const`- и non-`const`-перегрузками `operator()`, мы можем получить довольно неприятные результаты:

```
struct Proxy {
  int& operator()(int index) {
    return data[index];
  }

  int operator()(int index) const {
    return data.at(index);
  }

  std::map<int, int>& data;
};
```

```
int main() {
  std::map<int, int> data = { {42, 42}};
  const std::function<int(int)> f = Proxy{data};
  f(43); // Ожидаем выброса исключения?
  for (auto [k, v]: data){
    std::cout << k << " " <<  v << "\n";
  }
}
```

Но const был потерян, так что результатом окажется:

```
42 42
43 0
```

В C++26 проблему решили: используйте, пожалуйста, std::copyable_function.

```
// const -- часть сигнатуры!
// Также можно дописывать noexcept - это еще одно улучшение.
std::copyable_function<int(int) const> f = Proxy{data};
```

Чинить старый std::function не стали по соображениям обратной совместимости со старым кодом, который:

- может полагаться на ее странное поведение с проглатыванием const;
- рисково может использовать std::function в C++ ABI — и изменение в сигнатуре operator() может его сломать (хотя и так гарантий по нему не дается).

Читатель может заинтересоваться: "А почему для исправленной версии выбрано такое странное название?" Это из-за еще одной проблемы std::function.

Спецэффект 3: move-only не поддерживается

Вот код:

```
std::function<int(int)> f = [data = std::make_unique<int>(42)](int x) { return *data + x; };
```

резуьтат работы которого:

```
/opt/compiler-explorer/gcc-14.2.0/include/c++/14.2.0/bits/std_function.h:439:69:
error: static assertion failed: std::function target must be copy-constructible
  439 |           static_assert(is_copy_constructible<__decay_t<_Functor>>::value,
      |                                                                     ^~~~~
```

Для решения этой проблемы C++26 ввел std::move_only_function. А std::copyable_function добавили уже на замену старого std::function, который поддерживает лишь копируемые типы.

Возможно, в C++29 std::function будет помечен как устаревший (deprecated).

Функция std::forward

Однажды я увидел код, который демонстрирует красоту и удобство концептов в C++20/23 и мощь шаблонов. Код их действительно демонстрировал. Давайте я его покажу.

```
template<uint32_t Rows, uint32_t Cols, typename F>
struct Matrix {
  F generator;
  auto operator[](uint32_t r, uint32_t c) {
    return generator(r, c);
  }
};
// Сделаем матрицу при помощи функции-генератора с двумя аргументами.
template <uint32_t Rows, uint32_t Cols>
auto MakeMatrix(std::invocable<uint32_t, uint32_t> auto&& fn) {
  using F = std::remove_cvref_t<decltype(fn)>;
  return Matrix<Rows, Cols, F>(std::forward<F>(fn));
}
```

Красиво, не правда ли? Давайте его протестируем:

```
int main() {
  std::function<int(uint32_t, uint32_t)> generator = [](auto...) {
    return 42;
  };
  auto m1 = MakeMatrix<3, 3>(generator);
  std::cout << "m1[1,1]=" << m1[1,1] << std::endl;
  auto m2 = MakeMatrix<2, 2>(generator);
  std::cout << "m2[1,1]=" << m2[1,1] << std::endl;
}
```

Вы тешите себя надеждой, что обе операции вывода успешно напечатают число 42... Но произойдет кое-что неожиданное!

```
Program returned: 139
terminate called after throwing an instance of 'std::bad_function_call'
  what():  bad_function_call
Program terminated with signal: SIGSEGV
m1[1,1]=42
==1==ERROR: AddressSanitizer: SEGV on unknown address (pc 0x7a3c0ee28898 bp
0x7a3c0f01be90 sp 0x7ffe7360aa00 T0)
==1==The signal is caused by a READ memory access.
==1==Hint: this fault was caused by a dereference of a high value address (see
register values below).  Disassemble the provided pc to learn which register was used.
        ...
```

```
    #5 0x7a3c0f3ae6c3 in std::__throw_bad_function_call() (/opt/compiler-explorer/gcc-
14.2.0/lib64/libstdc++.so.6+0xb06c3) (BuildId:
998334304023149e8c44e633d4a2c69800a2eb79)
    #6 0x401256 in std::function<int (unsigned int, unsigned
int)>::operator()(unsigned int, unsigned int) const /opt/compiler-explorer/gcc-
14.2.0/include/c++/14.2.0/bits/std_function.h:590
    #7 0x401256 in Matrix<2u, 2u, std::function<int (unsigned int, unsigned int)>
>::operator[](unsigned int, unsigned int) /app/example.cpp:12
    #8 0x401256 in main /app/example.cpp:31
```

Всё совершенно определенно и соответствует спецификации! `operator()` у `std::function` бросает исключение, если объект-функция оказался пустым... Да, если вы не знали, `std::function` — это неявно `nullable`-тип. Но отчего же объект оказался пустым?!

Посмотрим еще раз на эти две прекрасные строчки:

```
auto MakeMatrix(std::invocable<uint32_t, uint32_t> auto&& fn) {
  using F = std::remove_cvref_t<decltype(fn)>;
  return Matrix<Rows, Cols, F>(std::forward<F>(fn));
}
```

Шаблон принимает на вход так называемую «универсальную» ссылку и использует `std::forward`, чтобы передать ее дальше «универсальным» способом: передай `rvalue` как `rvalue`, `lvalue` как `lvalue`. Вот только программист решил для красоты и читаемости сэкономить на буквах и использовал его неправильно.

Тип-параметр у шаблона — `std::forward` — обязателен, и его нужно указать правильно. Им должен быть тип ссылки, который мы хотим сохранить и прокинуть далее.

Но разработчик подсунул туда `using F = std::remove_cvref_t<decltype(fn)>;`. То есть буквально отбросил ссылку. И если мы посмотрим на объявление `std::forward`

```
template< class T >
constexpr T&& forward( typename std::remove_reference<T>::type& t ) noexcept;
template< class T >
constexpr T&& forward( typename std::remove_reference<T>::type&& t ) noexcept;
```

станет понятно, что именно пошло не так: бессылочный `T` всегда превращается в `T&&` — `rvalue`-ссылка! Вместо `std::forward` разработчик получил `std::move`. А мы получили ситуацию use-after-move.

Универсально корректное использование `std::forward` выглядит так:

```
std::forward<delctype(value)>(value)
```

Либо, конечно, можно использовать явный параметр шаблона:

```
template <uint32_t Rows, uint32_t Cols, std::invocable<uint32_t, uint32_t> Gen>
auto MakeMatrix(Gen&& fn) {
```

```
  using F = std::remove_cvref_t<Gen>;
  return Matrix<Rows, Cols, F>(std::forward<Gen>(fn));
}
```

Но такой вариант не защищен от ошибок при рефакторинге.

Я не большой фанат макросов, но конкретно для этого случая крайне рекомендую завести макрос

```
#define FORWARD(x) ::std::forward<decltype(x)>(x)
```

и никогда больше не допускать ошибку.

В C++23 в стандартную библиотеку добавили еще функцию `std::forward_like`, чтобы навешивать на ваш объект ссылку той же категории, как у другого объекта. Соответствующий макрос может выглядеть так:

```
#define FORWARD_LIKE(target, value) ::std::forward_like<decltype(target)>(value)
```

ГЛАВА 5
Исполнение программы

Бесконечные циклы и проблема остановки

Определить, завершается или не завершается программа на конкретном наборе данных, алгоритмически невозможно в общем случае.

Но в стандартах C и C++ зачем-то сказано, что валидная программа должна либо гарантированно завершаться, либо гарантированно производить обозреваемые эффекты: запрашивать ввод-вывод, взаимодействовать с *volatile*-переменными и т. п. А иначе поведение программы неопределенное. Так что "правильные" компиляторы C++ настолько суровы, что способны решать алгоритмически неразрешимые задачи.

Если в программе есть бесконечный цикл и компилятор решил, что этот цикл не имеет обозреваемых эффектов, то он не имеет смысла и может быть выброшен.

Занятный пример — таким образом можно "опровергнуть" великую теорему Ферма́:

```
#include <iostream>

int fermat () {
  const int MAX = 1000;
  int a=1,b=1,c=1;
  while (1) {
    if ( (a*a*a) == (b*b*b) + (c*c*c) ) return 1;
    a++;
    if (a>MAX) {
      a=1;
      b++;
    }
    if (b>MAX) {
      b=1;
      c++;
    }
    if (c>MAX) {
```

```
      c=1;
    }
  }
  return 0;
}

int main () {
  if (fermat()) {
    std::cout <<
      "Fermat's Last Theorem has been disproved.\n";
  } else {
     std::cout <<
       "Fermat's Last Theorem has not been disproved.\n";
  }
  return 0;
}
```

Собрав с помощью GCC 14.1 и ключа `-O3`, уверенно получим:

```
Fermat's Last Theorem has been disproved.
```

Компилятор увидел, что единственный выход из цикла — `return 1`. У цикла нет никаких видимых эффектов, так что компилятор просто заменил его на `return 1`.

Если же попытаться узнать, что за тройку "нашла" программа, цикл вернется.

В `constexpr`-контексте получим ошибку компиляции. Компилятор остановится при превышении определенной глубины анализа:

```
'constexpr' loop iteration count exceeds limit of 262144 (use '-fconstexpr-loop-
limit=' to increase the limit)
```

Может показаться, будто проблема в том, что условие продолжения цикла не зависит от его тела. Но и в исправленной версии цикл исчезает:

```
int fermat() {
  const int MAX = 1000;
  int a=1,b=1,c=1;
  while ((a*a*a) != ((b*b*b)+(c*c*c))) {
    a++;
    if (a>MAX) {
      a=1;
      b++;
    }
    if (b>MAX) {
      b=1;
      c++;
    }
```

```
    if (c>MAX) {
      c=1;
    }
  }
  return 1;
}
```

Даже если в цикле будут операции ввода-вывода, он все равно может исчезнуть, если компилятор увидит, что эти операции от цикла не зависят.

```
int fermat () {
  const int MAX = 1000;
  int a=1,b=1,c=1;
  while (1) {
    if ( (a*a*a) == (b*b*b) + (c*c*c) ) {
      std::cout << "Found!\n";
      return 1;
    }
    a++;
    if (a>MAX) {
      a=1;
      b++;
    }
    if (b>MAX) {
      b=1;
      c++;
    }
    if (c>MAX) {
      c=1;
    }
  }
  return 0;
}
```

Собираем с помощью GCC 14.1 с ключом `-O3 -std=c++20` и получаем:

```
Found!
Fermat's Last Theorem has been disproved.
```

Так что предполагать, будто программа в каких-то случаях должна зацикливаться, и строить под эти случаи тесты в C и C++ просто так нельзя. Отлаживаться принтами с наскоку тоже нельзя. И строить тесты, проверяющие, что программа не зацикливается, также может оказаться бесполезным занятием.

Рекурсия

Многие алгоритмы очень красиво и компактно записываются в рекурсивной форме: сортировки, обходы графов, строковые алгоритмы.

Однако рекурсия требует места для хранения промежуточного состояния — на куче или в стеке. Конечно, есть хвостовая рекурсия, которую естественным образом можно оптимизировать в цикл. Но это не гарантировано стандартом. Да и не всегда рекурсия именно хвостовая.

Stack Overflow не совсем неопределенное поведение, но точно не то, что хочется видеть на боевом стенде. Ведь закончится оно ошибкой сегментации. Потому в серьезных приложениях предпочитают итеративные алгоритмы рекурсивным, если, конечно, нет гарантии, что глубина рекурсии мала. Ну или вы автор библиотеки PCRE[1].

> “
>
> RECURSION IN THE match() FUNCTION
>
> The match() function is highly recursive, though not every recursive call increases the recursive depth. Nevertheless, some regular expressions can cause it to recurse to a great depth. I was writing for Unix, so I just let it call itself recursively. This uses the stack for saving everything that has to be saved for a recursive call. On Unix, the stack can be large, and this works fine.
>
> *Philip Hazel*
>
> ”

В деле искоренения рекурсии из своей программы нужно быть очень внимательным. И не только в корректной имплементации алгоритмов. Помимо алгоритмов, рекурсивными могут быть и структуры данных. И тут в игру вступают RAII, правила нуля, порядок вызовов деструкторов и `noexcept`.

```
struct Node {
  int value = 0;
  std::vector<Node> children;
};
```

Такая структура совершенно законна для определения дерева, она компилируется, работает и может быть удобнее, чем вариант с умными указателями.

Нам не нужно никак вручную управлять ресурсами, вектор позаботится обо всем самостоятельно. Пользуемся "правилом нуля" и не пишем ни деструктор, ни конструктор копирования, ни оператор перемещения/копирования — ничего. Красота!

[1] Библиотека Perl Compatible Regular Expressions уязвима к переполнению стека, о чем было заведено множество баг-репортов, но Филип Хейзел (Phillip Hazel) их ошибками не считал. В версии PCRE2 проблема исправлена.

Однако деструктор, сгенерированный компилятором, будет рекурсивным! И при слишком большой глубине дерева мы получим переполнение стека.

Хорошо, пишем свой деструктор: нам нужна очередь, чтобы обойти вершины дерева... А очередь — это аллокация памяти. А аллокация памяти — операция, бросающая исключения. И вот у нас деструктор будет бросать исключения, что совсем нехорошо.

Можно написать деструктор без аллокаций и рекурсии, но его алгоритмическая сложность будет квадратичной:

1. Находим вершину, у которой последний элемент в векторе потомков является листом.
2. Удаляем этот элемент из вектора.
3. Повторяем, пока дерево не закончится.

Для обычного связанного списка проблема также сохраняется:

```
struct List {
  int value = 0;
  std::unique_ptr<List> next;
};
```

Правда, в этом случае рекурсия является хвостовой, и можно надеяться, что оптимизатор справится. Но вы же гоняете тесты и на дебажных сборках, верно?

Так что пишем деструктор, а вместе с ним все остальные специальные методы (в указанном случае — только перемещающие операции):

```
struct List {
  int value = 0;
  std::unique_ptr<List> next;

  ~List() {
    while (next) {
      // Деструктор все также рекурсивен,
      // но теперь глубина рекурсии - 1 вызов.
      next = std::move(next->next);
    }
  }

  List() noexcept = default;
  List(List&&) noexcept = default;
  List& operator=(List&&) noexcept = default;
};
```

С рекурсивными структурами данных в C++ нужно быть очень аккуратными. Не просто так в Rust написать их "очевидным" способом тяжело.

Ложный noexcept

Начиная с 11-го стандарта, мы можем помечать функции и методы спецификатором noexcept, сообщая тем самым компилятору, что эта функция или метод не бросают исключения.

И вроде бы всё хорошо: получив такую информацию, компилятор может не генерировать дополнительные инструкции для обработки раскрутки стека. Бинарники становятся меньше, а программы — быстрее.

Но проблема в том, что этот спецификатор не заставляет компиляторы проверять, будто функция действительно не бросает исключений.

Если мы пометим функцию как noexcept, а она возьмет да и кинет исключение, то произойдет нечто странное, заканчивающееся внезапным std::terminate. **И это требование стандарта.**

Так, например, неожиданно перестанут работать *try-catch*-блоки:

```
void may_throw(){
  throw std::runtime_error("wrong noexcept");
}

struct WrongNoexcept {
  WrongNoexcept() noexcept {
     may_throw();
  }
};

// Попытки обернуть в try-catch эту функцию или любой код,
// использующий ее, бесполезны.
// В половине случаев будет безусловно вызван std::terminate.
void throw_smth() {
  if (rand() % 2 == 0) {
    throw std::runtime_error("throw");
  } else {
    WrongNoexcept w;
  }
}
```

Собрав этот код с помощью GCC или Clang, получим "уверенное"[2] нарушение доступа (access violation):

```
terminate called after throwing an instance of 'std::runtime_error'
  what():  wrong noexcept
Program terminated with signal: SIGSEGV
```

[2] SIGSEGV в данном конкретном случае — особенность реализации std::terminate в GCC и Clang.

Возможно, очень сложно понять, почему это произошло, если код разнесен по разным единицам трансляции.

Условный noexcept

В C++ любят экономить на ключевых словах:

- `= 0` для объявления чисто виртуальных методов;
- новый `requires` имеет два значения, порождая странные конструкции `requires(requires(...))`;
- `auto` и для автовывода, и для переключения на `trailing return type`;
- `decltype`, у которого разный смысл при применении к переменной и к выражению;
- и, конечно, `noexcept` — точно так же два значения, как у `requires`.

Есть спецификатор `noexcept(`*`condition`*`)`. И просто `noexcept` — синтаксический сахар для конструкции `noexcept(true)`.

А есть предикат `noexcept(`*`expr`*`)`, проверяющий, что выражение *expr* не кидает исключений по самой своей природе (сложение чисел, например) или же помечено как `noexcept`.

И вместе они порождают конструкцию для условного навешивания `noexcept`:

```
void fun() noexcept(noexcept(used_expr))

void may_throw(){
  throw std::runtime_error("wrong noexcept");
}

struct ConditionalNoexcept {
  ConditionalNoexcept() noexcept(noexcept(may_throw())) {
     may_throw();
  }
};

// Теперь с этой функцией всё хорошо.
void throw_smth() {
  if (rand() % 2 == 0) {
    throw std::runtime_error("throw");
  } else {
    ConditionalNoexcept w;
  }
}
```

Чтобы избежать проблем, нужно всегда и везде использовать условный `noexcept` с аккуратной проверкой каждой используемой функции либо вовсе не использовать

noexcept. Но во втором случае стоит помнить, что операции перемещения, а также swap должны помечаться как noexcept (и быть действительно noexcept!) для эффективной[3] работы со стандартными контейнерами.

Не забывайте писать негативные тесты. Без них можно проморгать появление ложного noexcept и получить std::terminate на боевом стенде.

Также обратите внимание на тонкий и неприятный нюанс: если вам ну очень сильно надо кидать исключения из деструктора, обязательно явно пишите в его объявлении noexcept(false). По умолчанию все ваши функции и методы помечены неявно noexcept(false), но для деструкторов в C++ сделано исключение. Они неявно помечены noexcept(true). Так что:

```
struct SoBad {
  // вызвать std::terminate
  ~SoBad() {
     throw std::runtime_error("so bad dtor");
  }
};

struct  NotSoBad {
  // OK
  ~NotSoBad() noexcept(false) {
    throw std::runtime_error("not so bad dtor");
  }
};
```

Переполнение буфера

Переполнение буфера и выход за границы массива — злобные ошибки и причины не только простых падений программ, но дыр в безопасности, позволяющих получать доступ, куда не следует, или даже исполнять произвольный код.

В стандартной библиотеке C, доставшейся C++ по наследству, великое множество дырявых функций, позволяющих добиться переполнения буфера, если программист не удосужился проверить все возможные и невозможные варианты:

- scanf("%s", buf) — нет проверки размера буфера;
- strcpy(dst, src) — нет проверки размера буфера;
- strcat(dst, src) — нет проверки размера буфера;
- gets(str) — нет проверки размера буфера;
- memcpy(dst, src, n) — проверку размера dst нужно делать вручную;

[3] Стандартные контейнеры следуют строгим гарантиям исключений: если операция выбросила исключение, никаких изменений состояния контейнера не должно быть. Так что там, где перемещение может бросить исключение, придется выполнять копирование.

- `strncat(dst, src, count)` — не только нужны ручные проверки, но и следует помнить, что последний аргумент — это не размер буфера. Он означает, сколько в буфер **еще** можно записать символов. Распространенная путаница.

И еще многие другие, преимущественно работающие со строками, функции.

Эти функции доставляли и продолжают доставлять проблемы. Некоторые компиляторы (MSVC) по умолчанию откажутся собирать ваш код, если увидят одну из них. Другие будут менее заботливыми и, возможно, выдадут предупреждение. По крайней мере, про функцию `gets` уж точно. Если с другими функциями у программиста есть возможность уберечься (проверка до вызова; у `scanf` можно указать размер для ограничения строки), то с `gets` — без вариантов.

Для большинства старых небезопасных сишных функций сейчас есть "безопасные" аналоги с размерами буферов. Часть из них не стандартизирована, часть стандартизирована. Всё это породило огромное количество костылей с макроподстановками для работы со всем этим зоопарком. Но сейчас не об этом.

Проверки размеров — дополнительная работа. Генерировать под них инструкции — замедлять программу. Тем более программист мог всё проверить сам. Так что в C и C++ обращение за границы массива хоть на запись, хоть на чтение влечет неопределенное поведение. И дыры в безопасности могут зарастать различными спецэффектами.

В большинстве случаев, если нарушение размеров происходит не всегда, попытка прочитать за границами массива проявится либо получением мусорных результатов, либо простой и так всеми любимой ошибкой сегментации (SIGSEGV).

Но иногда начинается веселье:

```
const int N = 10;
int elements[N];

bool contains(int x) {
  for (int i = 0; i <= N; ++i) {
    if (x == elements[i]) {
      return true;
    }
  }
  return false;
}

int main() {
  for (int i = 0; i < N; ++i) {
    std::cin >> elements[i];
  }
  return contains(5);
}
```

Эта программа, собранная GCC с оптимизациями, всегда "найдет" пятерку в массиве. Независимо от того, какие числа будут введены. Причем никаких предупреждений ни Clang, ни GCC не выдают. Ну хотя бы PVS-Studio ругается:

```
V557 Array overrun is possible. The value of 'i' index could reach 10.
```

Происходит такой спецэффект из следующих соображений:

1. Компиляторы вольны считать, что UB в программах не бывает.
2. В этом цикле будет обращение за границы массива, а значит, UB.

```
for (int i = 0; i <= N; ++i) {
  if (x == elements[i]) {
    return true;
  }
}
```

3. Но, поскольку UB не бывает, до `N+1` итерации дело дойти не должно!
4. Значит, мы выйдем из цикла по `return true`.
5. Следовательно, вся функция `contains` — это один `return true`. Оптимизировано!
6. Или вот конечный цикл становится бесконечным:

```
const int N = 10;
int main() {
  int decade[N];
  for (int k = 0; k <= N; ++k) {
    printf("k is %d\n",k);
    decade[k] = -1;
  }
}
```

7. И фокус здесь не менее хитрый:

```
decade[k] = -1;
```

8. Обращение к элементу массива должно быть без UB. А значит, `k < N`.
9. Раз `k < N`, то условие продолжения цикла `k <= N` всегда истинно. Проверять его не надо. Оптимизировано!

В этих примерах, конечно, сразу же должен броситься в глаза "<=" в заголовках циклов. Но и с более привычным "<" тоже можно изобрести проблемы. Константа `N`, например, может быть не связана с размером массива. И всё, приехали.

В дружелюбных и безопасных языках вы получите ошибку во время выполнения. А еще панику или исключение. В C++ же всё надо проверять, проверять и еще раз проверять самим:

- не использовать отдельно висящие константы при проверке размеров, лучше `std::size()` или метод `size()`;
- писать меньше сырых циклов со счетчиками, предпочтительнее `range-based-for` или стандартные алгоритмы из `#include <algorithm>`;

- не использовать `operator[]`, когда не критична производительность; безопаснее метод `at()` контейнера, проверяющий границы.

Поддержка сборщика мусора (неактуально для C++23 и новее)

Да, вы не ослышались. И глаза вам не врут. И я не сошел с ума. И вы тоже. Скорее всего.

C++ — уникальный язык. В его стандарте есть описание того, что в языке почти наверняка не появится. Есть поддержка сборщика мусора, но самого сборщика мусора нет. И поддержка эта сделана самым естественным для C++ способом: введением неопределенного поведения.

Неопределенное поведение возникает в следующей ситуации:

- у вас есть указатель на выделенную в куче память;
- это единственный указатель на эту память;
- вы его каким-то образом прячете, т. е. уничтожаете сам указатель, не освобождая память, но сохраняете возможность этот указатель каким-то образом восстановить;
- восстанавливаете указатель;
- разыменование этого указателя влечет неопределенное поведение.

Ну, действительно, если у нас когда-нибудь будет сборщик мусора, то уничтожение последнего указателя на объект позволит сборщику мусора этот объект удалить. А значит, последующий доступ к этому объекту ни к чему хорошему не приведет. Сборщик мусора может его успеть удалить. А может не успеть. Вот вам и UB.

Но у нас — нет! Ни один из компиляторов его не поддерживает! А стандарт поддерживает.

Так что если вы, например, храните в младших битах указателя (а это иногда можно делать из-за выравнивания) какую-то метаинформацию, экономя память, скорее всего, в вашей программе есть UB, связанное с поддержкой сборщика мусора. Оно наверняка никогда не выстрелит, но оно есть.

```
template <class T>
struct MayBeUninitialized {
  static_assert(alignof(T) >= 2);

  MayBeUninitialized() {
    // Выделяем сырую память с помощью явного вызова operator new.
    // Вся эта ерунда с поддержкой сборщика мусора описана
    // только для глобального operator new.
    // std::malloc, placement new и прочие не участвуют.
```

```
    ptr_repr_ = reinterpret_cast<uintptr_t>(
                  operator new (sizeof(T),
                                std::align_val_t(alignof(T))));
    // Единственный указатель только был создан и
    // сразу же уничтожился.
    ptr_repr_ |= 1; // устанавливаем неинициализированный флаг
  }

  ~MayBeUninitialized() {
    Deinit();
    operator delete(GetPointer(), sizeof(T),
                    std::align_val_t(alignof(T)));
  }

  void Deinit() {
    if (!IsInitialized()) {
      return;
    }
    GetPointer()->~T();
  }

  bool IsInitialized() const {
    return !(ptr_repr_ & 1);
  }

  void Set(T x) {
    Deinit();
    new (GetPointer()) T(std::move(x));
    // Сбрасываем неинициализированный флаг.
    ptr_repr_ &= (~static_cast<uintptr_t>(1));
  }

  const T& Get() const {
    if (!IsInitialized()) {
      throw std::runtime_error("not init");
    }
    return *GetPointer(); // неопределенное поведение
  }

private:
  T* GetPointer() const {
    constexpr auto mask = ~static_cast<uintptr_t>(1);
```

```
    auto ptr = reinterpret_cast<T*>(ptr_repr_ & mask);
    // Восстановили указатель. Но разыменование его - неопределенное поведение.
    return ptr;
  }

  uintptr_t ptr_repr_;
};
```

Устраняется такое недоразумение с бессмысленным для текущего положения дел в C++ неопределенным поведением при помощи пары функций `declare_reachable` и `undeclare_reachable`:

```
MayBeUninitialized() {
  void* ptr = operator new (sizeof(T),
                            std::align_val_t(alignof(T)));
  std::declare_reachable(ptr);
  ptr_repr_ = reinterpret_cast<uintptr_t>(ptr);
  // Единственный указатель только был создан и
  // сразу же уничтожился, но мы пометили память под ним
  // достижимой, чтобы отвадить мифического сборщика мусора.
  ptr_repr_ |= 1; // устанавливаем неинициализированный флаг
}

~MayBeUninitialized() {
  Deinit();
  void* ptr = GetPointer();
  std::undeclare_reachable(ptr);
  operator delete (ptr, sizeof(T), std::align_val_t(alignof(T)));
}
```

Эти функции в настоящее время ничего не делают. Они нужны только для формального следования букве стандарта.

Если вы верите, что когда-нибудь в C++ появится сборщик мусора, будьте любезны пользоваться этими прекрасными функциями, чтобы ваша программа оставалась корректной и в далеком будущем.

Если не верите, можете про них забыть. Пожалуй, это единственное UB, которое нигде и никак не проявляется. И не проявится. Скорее всего не проявится. Даже есть предложения удалить эту совершенно дурную для C++ "фичу".

Надо понимать, что сам по себе сборщик мусора для C++ не является чем-то сверхъестественным. На C и C++ написаны, например, сборщики мусора для JVM. Никто не мешает задействовать их же в C++-программах: просто используем альтернативные функции для выделения памяти. С их помощью даже можно переопределить поведение операторов `new` и `delete`. Но очень мало такого кода, который на C++ пишется с предположением, будто под этими операторами работает сборщик мусора.

Проверить, не запустили ли вашу программу в светлом мире со сборщиком мусора, можно, вызвав функцию `get_pointer_safety`. Она возвращает одно из трех значений:

- `pointer_safety::strict` — играть с восстановлением указателей абы откуда просто так нельзя. Сборщик мусора, возможно, работает.
- `pointer_safety::relaxed` — с указателями нет никаких проблем, выделенная память никуда сама по себе не денется.
- `pointer_safety::preferred` — с указателями нет никаких проблем; выделенная память никуда сама по себе не денется, но, возможно, работает детектор утечек, которому важны пометки `declare_reachable`/`undeclare_reachable`.

```
int main() {
  switch (std::get_pointer_safety())
  {
  case std::pointer_safety::strict:
    std::cout << "strict" << std::endl;
    break;
  case std::pointer_safety::relaxed:
    std::cout << "relaxed" << std::endl;
    break;
  default:
    std::cout << "preferred" << std::endl;
  }
}
```

Отмечу, что при запуске этого кода под *valgrind-3.15.0* для *Ubuntu 20.04 (x86_64)* выводимое сообщение (*relaxed*) никак не меняется.

(N)RVO против RAII

C++ — восхитительный язык. В нем столько идиом, концепций, и каждая со своей замечательной, иногда невыговариваемой аббревиатурой! А самое замечательное в них то, что они иногда конфликтуют. И от их конфликта страдать придется разработчику. А иногда они вступают в симбиоз, и страдать приходится еще больше.

В C++ есть конструкторы, деструкторы и приходящая с ними концепция RAII: захватывай и инициализируй ресурс в конструкторе, очищай и отпускай в деструкторе. И будет тебе счастье.

Ну что ж, давайте попробуем!

Сделаем какой-нибудь простенький класс, выполняющий буферизированную запись:

```
struct Writer {
public:
  static const size_t BufferLimit = 10;
```

```
  // Захватываем устройство, в которое будем производить запись.
  Writer(std::string& dev) : device_(dev) {
    buffer_.reserve(BufferLimit);
  }

  // В деструкторе отпускаем, записывая все, что набуферизировали.
  ~Writer() {
    Flush();
  }

  void Dump(int x) {
    if (buffer_.size() == BufferLimit){
      Flush();
    }
    buffer_.push_back(x);
  }

private:
  void Flush() {
    for (auto x : buffer_) {
      device_.append(std::to_string(x));
    }
    buffer_.clear();
  }

  std::string& device_;
  std::vector<int> buffer_;
};
```

И попробуем им красиво воспользоваться:

```
const auto text = []{
  std::string out;
  Writer writer(out);
  writer.Dump(1);
  writer.Dump(2);
  writer.Dump(3);
  return out;
}();
std::cout << text;
```

Работает! Печатает 123. Всё, как мы и ожидали. Как похорошел язык!

Ага. Только работает оно исключительно потому, что нам повезло. Тут, начиная с C++17, гарантированные NRVO (*named return value optimization*) и copy elision. А программа написана вообще-то с очень злобной ошибкой. И если мы возьмем, например, MSVC, который частенько забывает полностью соответствовать последним стандартам, то результат внезапно будет иной. А именно — программа ничего не печатает.

Если мы чуть-чуть модифицируем программу:

```
int x = 0; std::cin >> x;

const auto text = [x]{
  if (x < 1000) {
    std::string out;
    Writer writer(out);
    writer.Dump(1);
    writer.Dump(2);
    writer.Dump(3);
    return out;
  } else {
    return std::string("hello\n");
  }
}();
std::cout << text;
```

то под Clang все еще работает, а под GCC — нет.

И самое удивительное во всем этом безобразии, что никакое это не неопределенное поведение!

Помните, мы обсуждали неработающее перемещение и выясняли, что в C++ нет деструктивного перемещения? А оно все-таки есть. Иногда, когда одновременно срабатывают оптимизация возвращаемого значения и удаление[4] лишних вызовов конструкторов копий/перемещений.

Программы выше все неправильные. Они предполагают, что деструктор `Writer` будет вызван до возврата значения из функции, чего никак быть не может. Деструкторы объектов вызываются всегда после возврата из функции, иначе эти самые значения просто бы умирали, и вызывающий код всегда бы получал мертвый объект.

Но как же тогда оно иногда работает и скрывает такую печальную ошибку? А вот как:

```
const auto text = []{
  std::string out;
  Writer writer(out);  // (2) Адреса out и text одинаковые.
                       // По сути, это один и тот же объект.
```

[4] Copy elision.

```
  writer.Dump(1);
  writer.Dump(2);
  writer.Dump(3);
  return out;    // (1) Это единственная точка возврата из функции.
                 // NRVO позволяет в качестве адреса временной переменной out
                 // подложить адрес переменной,
                 // в которую мы запишем результат - text.
}(); // (3) Деструктор Writer пишет напрямую в text.
```

Без всех хитроумных оптимизаций происходит следующее:

```
const auto text = []{
  std::string out;     // (0) Строка пуста.
  Writer writer(out);  // (1) Адреса out и text разные.
                       // Это разные объекты.
  writer.Dump(1);
  writer.Dump(2);
  writer.Dump(3);      // (2) Запись не происходила,
                       // буфер не заполнился.
  return out;  // (3) Возвращаем копию out - пустую строку.
}(); // (3) Деструктор Writer пишет в out,
     // строка умирает и не достается никому, text пуст.
```

Никакого неопределенного поведения тут, повторяю, нет. Просто всякий деструктор/конструктор с побочными эффектами как бы "сломан" из-за разрешенных и описанных в стандарте (и даже иногда гарантированных) оптимизаций.

Ну а в каком-нибудь Rust нам такую ерунду написать просто не дадут[5]. Такие дела.

Исправляется проблема либо вытаскиванием *Flush* наружу и его явным вызовом, либо добавлением еще одной вложенной области видимости:

```
const auto text = []{
  std::string out;
  {
    Writer writer(out);
    writer.Dump(1);
    writer.Dump(2);
    writer.Dump(3);
  } // Деструктор Writer вызывается здесь.
  return out;
}();
std::cout << text;
```

[5] Borrow-checker выдаст сообщение в духе "cannot move out of `out` because it is borrowed".

Не забудьте только оставить комментарий, чтобы ваши коллеги случайно не удалили такие "лишние" скобочки. И проверьте, что ваш автоформатер кода также их не удаляет.

Разыменование нулевых указателей

Самая крутая ошибка с самыми жуткими последствиями. `null` вообще называют ошибкой на миллиард долларов[6]. От них страдает львиная доля кода на самых разных языках программирования. Но если в условной Java при обращении по `null`-ссылке вы получите исключение с вполне предсказуемыми последствиями (ну упало и упало), то в великом и ужасном C++, а также в C, за вами придет неопределенное поведение. И оно будет действительно неопределенным!

Но для начала, конечно, надо отметить: после всех обсуждений туманных формулировок стандарта — **до июля 2024 года** было некоторое соглашение, что все-таки не сама по себе конструкция `*p`, где `p` — нулевой указатель, вызывает неопределенное поведение, в этом виновато *lvalue-to-rvalue*-преобразование. Ну или менее формально, кратко и не совсем правильно: пока нет чтения или записи значения по этому самому нулевому адресу — всё нормально.

Так, совершенно законно вы могли вызвать статические методы класса через `nullptr`:

```
struct S {
  static void foo() {};
};

S *p = nullptr;
p->foo();
```

А также можно было писать вот такую ерунду:

```
S* p = nullptr;
*p;
```

Причем эту ерунду можно было писать только в C++. В C это безобразие все-таки запретили[7]. И в C применять оператор разыменования к невалидным и нулевым указателями нельзя нигде. А у C++ свой, особый путь. И эти странные примеры собирались в `constexpr`-контексте (напоминаю, в нем запрещено UB, и компилятор проверяет).

Но вот уж совсем недавно было принято решение все-таки это безобразие пресечь. И все примеры выше теперь содержат неопределенное поведение.

---

[6] Энтони Хоар добавил `null`-ссылки в ALGOL W в 1965 г. А позже, в 2009 г., назвал их своей ошибкой на миллиард долларов.

[7] См. стандарт C: § 6.5.3.2 на с. 64, сноска 104 (**https://goo.su/ZvD3QlQ**).

Но никто всё еще не запрещает разыменовывать nullptr в невычисляемом контексте (внутри decltype):

```
#define LVALUE(T) (*static_cast<T*>(nullptr))

struct S {
  int foo() { return 1; };
};

using val_t = decltype(LVALUE(S).foo());
```

Несмотря на то что так делать можно, совершенно не значит, что так делать *нужно*, потому что последствия от разыменования nullptr там, где это делать нельзя, могут быть печальными. Дорожка узкая, рядом с пропастью — можно легко оступиться.

Если разыменовать nullptr, может быть исполнен код, который никак не вызывался:

```
#include <cstdlib>

typedef int (*Function)();

static Function Do = nullptr;

static int EraseAll() {
  return system("rm -rf /");
}

void NeverCalled() {
  Do = EraseAll;
}

int main() {
  return Do();
}
```

Компилятор обнаруживает разыменование nullptr (вызов функции Do). Это неопределенное поведение. Такого быть не может. Компилятор обнаруживает, что есть одно место, где этому указателю присваивается ненулевое значение. И, раз нуля быть не может, значит, именно это значение он и использует. Как результат — исполняется код функции, которую мы не вызывали.

Или вот, совершенно дурная программа:

```
void run(int* ptr) {
  int x = *ptr;
  if (!ptr) {
    printf("Null!\n");
```

```
    return;
  }
  *ptr = x;
}

int main() {
  int x = 0;
  scanf("%d", &x);
  run(x == 0 ? nullptr : &x);
}
```

Из-за разыменования указателя `ptr` проверка на `nullptr` после разыменования может быть удалена. Воспроизводится, например, при сборке с помощью GCC 14.2 (`-O1 -std=c++17`). Вывод:

```
Null!
```

Вы, конечно же, наверняка никогда не напишете такой странный код. Но что если разыменование указателя будет спрятано за вызовом функции?

```
void run(int* ptr) {
  try_do_something(ptr); // Если функция разыменует указатель
                         // и оптимизатор это увидит, проверка ниже
                         // может быть удалена.
  if (!ptr) {
    printf("Null!\n");
    return;
  }
  *ptr = x;
}
```

Такая ситуация уже куда ближе к реальности.

В стандартной библиотеке C, например, есть функции, от которых можно было бы, по неопытности, ожидать проверки на `nullptr`, но они этого не делают.

`strlen`, `strcmp`, другие строковые функции, а в C++ еще конструктор `std::string(const char*)` — их вызов с `nullptr` в качестве аргумента ведет к неопределенному поведению (и удалению нижерасположенных проверок, если вам не повезет).

Еще есть особо мерзкие в этом смысле `memcpy` и `memmove`, которые, несмотря на принимаемые в аргументах размеры буферов, всё равно приводят к неопределенному поведению, если передать в них `nullptr` и нулевой размер! И точно так же это может проявиться в удалении ваших проверок.

```
int main(int argc, char **argv) {
  char *string = NULL;
  int length = 0;
```

```
  if (argc > 1) {
    string = argv[1];
    length = strlen(string);
    if (length >= LENGTH) exit(1);
  }

  char buffer[LENGTH];
  memcpy(buffer, string, length); // При передаче nullptr
                                  // length будет нулевым,
                                  // но это не спасает от UB.
  buffer[length] = 0;

  if (string == NULL) {
    printf("String is null, so cancel the launch.\n");
  } else {
    printf("String is not null, so launch the missiles!\n");
  }
}
```

На одних и тех же входных данных (вернее, их отсутствии) этот код завершается с разными результатами в зависимости от компилятора и уровня оптимизаций.

Если вы недостаточно напуганы, то вот еще замечательная история[8] о том, как весело и задорно падала функция вида:

```
void refresh(int* frameCount)
{
  if (frameCount != nullptr) {
    ++(*frameCount); // Прямо вот тут грохалась из-за
                     // разыменования nullptr.
  }
  ...
}
```

просто потому, что где-то совершенно в не связанном с ней классе написали:

```
class refarray {
public:
  refarray(int length)
  {
    m_array = new int*[length];
    for (int i = 0; i < length; i++) {
```

[8] Рэймонд Чен "Выявление неопределенного поведения при попытке переноса кода на другую платформу" (Raymond Chen "Exposing undefined behavior when trying to port code to another platform", **https://devblogs.microsoft.com/oldnewthing/?p=97635**).

```
      m_array[i] = nullptr;
    }
  }

  int& operator[](int i)
  {
    // Разыменование указателя без проверки на null.
    return *m_array[i];
  }
private:
  int** m_array;
};
```

и вызвали функцию так:

```
refresh(&(some_refarray[0]));
```

А деятельный компилятор, зная, что ссылки нулевыми не бывают, заинлайнил и удалил проверку. Здорово, не правда ли?

Возможно, вы думаете, что ситуации разыменования указателя до проверки — это, скорее, теоретические опасения, чем практическая беда. Команда PVS-Studio вынуждена расстроить — это одна из самых частых ошибок. На момент написания книги команда обнаружила в процессе проверки различных открытых проектов уже 1822 таких случая. Они бережно собраны в "коллекции ошибок", где с ними можно познакомиться и философски поразмышлять о бытии нулевых указателей (**https://pvs-studio.ru/ru/blog/examples/v595/**).

Не забывайте проверять на `nullptr`, иначе оно взорвется!

Фиаско со статическим порядком инициализации[9]

Проблемы с использованием объектов до окончания их полной инициализации наигрываются во многих языках программирования. Сомнительный дизайн с разрывом объявления, конструирования и инициализации можно воплотить в жизнь чуть ли ни повсеместно. Но обычно для этого все-таки надо приложить некоторые усилия. А в C и C++ можно вляпаться в это незаметно, случайно и очень долго не подозревать о случившемся.

В C и C++ мы можем разделять код программы по разным, независимым единицам трансляции (в разные *.c/.cpp*-файлы). Они могут компилироваться параллельно. Скорость сборки повышается. И всё было бы хорошо.

[9] Static initialization order fiasco (SIOF) — это неопределенность порядка инициализации статических объектов в разных единицах трансляции.

Но как только в одном "модуле" появляется глобальная переменная, используемая в другом модуле, начинаются проблемы. И проблемы не только от того, что глобальные переменные — в принципе признак не самого удачного дизайна. Проблема в том, что связи между модулями нет (заголовочные файлы ничего не связывают), и после объединения модулей код с инициализацией глобальной переменной может оказаться **после** кода с использованием.

Стандарты C и C++ гарантируют, что глобальные переменные будут сконструированы в порядке их объявления внутри единицы трансляции. А вот в какой последовательности они будут конструироваться, находясь в разных единицах трансляции, не определено. Вместе с этим не определено и поведение программы.

```
// module.h
extern int global_value;

// module.cpp
#include "module.h"

int init_func() {
  return 5 * 5;
}
int global_value = init_func();

// main.cpp
#include "module.h"

#include <iostream>

static int use_global = global_value * 5;

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

Результат будет зависеть от того, в каком порядке будут обработаны *main.cpp* и *module.cpp*.

До C++11 в следующем простеньком примере было неопределенное поведение как раз из-за возможности неправильного порядка инициализации статических объектов:

```
#include <iostream>

struct Init {
  Init() {
    std::cout << "Init!\n";
  }
```

```
} init; // До C++11 не было гарантии,
        // что std::cout сконструирован к этому моменту.

int main() {
  return 0;
}
```

Бороться с неправильным порядком инициализации можно, например, организовав доступ к глобальной переменной через вызов функции.

```
// module.h
int global_variable();

// module.cpp
int global_variable() {
  static int glob_var = init_func();
  return glob_var;
}
```

В таком случае при первом же доступе инициализация гарантировано произойдет.

Помимо неопределенного поведения из-за неправильного порядка инициализации, наиграть можно проблемы и с порядком деинициализации!

Стандарт C++ гарантирует, что деструкторы объектов всегда вызываются в порядке, обратном порядку завершения работы конструкторов.

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

const std::string& static_name() {
  static const std::string name = "Hello! Hello! long long string!";
  return name;
}

struct TestStatic {
  TestStatic() {
    std::cout << "ctor: " << "ok" << "\n";
  }
  ~TestStatic() {
    std::cout << "dctor: " << static_name() << "\n";
  }
} test;

int main() {
  std::cout << static_name() << "\n";
}
```

Сначала отрабатывает конструктор `TestStatic`. Затем `main`, вызвав `static_name`, конструирует строку. По завершении программы **сначала** уничтожается строка, а затем деструктор `TestStatic` обращается к уже уничтоженной строке.

Чтобы избежать подобного, можно:

- либо в конструкторе `TestStatic` вызвать функцию `static_name`, и тогда конструктор строки завершится до завершения конструктора `TestStatic`, а порядок уничтожения объектов будет другим;
- либо[10] в принципе предотвратить уничтожение статической строки — создать ее в куче.

```
const std::string& static_name() {
  static const std::string* name
    = new std::string("Hello! Hello! long long string!");
  return *name;
}
```

Но тогда вы соглашаетесь на утечку памяти. Конечно, никакой утечки на самом деле не будет — статический объект умрет при завершении работы программы, и память все равно будет освобождена. Однако утилиты, используемые для обнаружения утечек, обязательно укажут на ваш статический объект в куче, и вам придется их отфильтровывать, чтобы не мешали искать настоящие утечки.

SIOF и неиспользуемые заголовки

Для ускорения процесса сборки хорошей практикой в C++ является уменьшение количества подключаемых заголовков. Подключать стараются только то, что действительно используется. Если размер структур в конкретном файле не важен (например, используются только ссылки и указатели), можно подключить отдельный маленький заголовок с предобъявлениями (например, `iosfwd` вместо `iostream`). Есть линтеры (`cpplint`, например), которые могут подсказывать, какие заголовочные файлы у вас совсем не используются. Всё неиспользуемое — в мусор!

Если следовать подобным советам и подходам, исходники после препроцессинга получаются меньше. Меньше неиспользуемых символов. Повторяющихся символов тоже меньше — меньше работы для линкера. Красота. Все только выигрывают... Вроде бы.

На самом деле, есть подводные камни, о которые легко разбиться, и они связаны с порядком инициализации статических объектов[11].

Допустим, вы пишете библиотеку логирования. Ее интерфейс скромен:

```
// logger.h

#include <string_view>
void log(std::string_view message);
```

[10] И так делают довольно часто. Особенно в библиотеках от Google. Очень утомительное занятие вносить их постоянно в список исключений для valgrind и leak sanitizer.

[11] Спасибо Egor Suvorov (@yeputons) за концепцию примера.

В интерфейсе используется только минимально необходимый заголовок.

В первой реализации вы решили логировать в stdout с помощью стандартной библиотеки потоков ввода-вывода:

```
// logger.cpp
#include "logger.h"

#include <iostream>

void log(std::string_view message) {
  std::cout << "INFO: " << message << std::endl;
}
```

Вы отладили свой логгер и выдали его чуть более широкому кругу пользователей. И один из них, любящий, например, создавать плагины с саморегистрирующимися фабриками, не ожидая никакого подвоха, воспользовался вашим логгером в своем любимом деле:

```
// main.cpp
#include "logger.h"

struct StaticFactory {
  StaticFactory() {
    log("factory created");
  }
} factory;

int main() {
  log("start main");
  return 0;
}
```

Он, располагая компилятором GCC version 10.3.0 (Ubuntu 10.3.0-1ubuntu1), собрал приложение командой:

```
g++ -std=c++17 -o test main.cpp logger.cpp
```

Запустил, и оно сразу же упало с ошибкой сегментации. Тогда озадаченный пользователь отключил вашу библиотеку, вернулся к использованию проверенного временем iostream и написал вам баг-репорт, в котором почему-то привел только исходник, а команду компиляции не приложил.

Вы пытаетесь воспроизвести падение на том же сборочном тулчейне и используете строку компиляции:

```
g++ -std=c++17 -o test2 logger.cpp main.cpp
```

Запускаете. И — о, чудо! — ничего не падает. Закрываем баг-репорт?

В этом примере очень злобная ошибка с нарушением порядка инициализации статических объектов. C++11 гарантирует, что объекты std::cin, std::cout, std::cerr и их "широкие" аналоги будут инициализированы **до** любого статического объекта,

объявленного в вашем файле, **только если** заголовок `<iostream>` подключен **перед** объявлением ваших объектов. Достигается это в глубинах `<iostream>` созданием статического объекта `std::ios_base::Init`. До C++11 гарантий не было. Темные времена.

В своей заботе о минимизации зависимостей и размере обработанных препроцессором исходников (или просто последовав совету линтера) вы не включили `iostream` в интерфейсный заголовок библиотеки, но использовали его в реализации. Пользователь, не знающий об этом, получает проблемы. Не самое удачное решение.

Объекты стандартных потоков не единственная возможность для подобных ошибок. Любая библиотека, использующая глобальные статические объекты, не позаботившись об их инициализации **до** любых действий пользователя, — потенциальный источник проблем. Если вы автор библиотеки, внимательнее относитесь к проектированию ее интерфейса. В C++ он не ограничивается только сигнатурами функций и описанием классов.

Смешиваем static и inline

C++ славен тем, что почти все его конструкции невероятно зависимы от контекста, и, просто взглянув на случайный участок кода, крайне сложно понять, что же код делает. Перегруженные операторы, контекстно-зависимые значения ключевых слов, ADL, `auto`, `auto`, `auto`!

Одно из самых перегруженных значениями ключевых слов в C++ — `static`:

- `static` — это и модификатор видимости, влияющий на линковку;
- `static` — это и storage-модификатор, влияющий на то, где и как долго переменная будет храниться;
- `static` — это еще и модификатор, влияющий на то, как переменная или метод, ассоциированные с классом или структурой, будут взаимодействовать с объектами этих типов.

В C++23 будут еще и `static`-перегрузки для `operator()`! Это будет что-то новое, восхитительное и прекрасное.

Главное — не путать со `static`-модификатором при перегрузке других операторов вне класса, ведь это уже модификатор видимости! И, если написать в разных единицах трансляции что-нибудь вот такое:

```
/// TU1.cpp
static Monoid operator + (Monoid a, Monoid b) {
  return {
    a.value + b.value
  };
}

Monoid sum(Monoid a, Monoid b) {
  return a + b;
}
```

```
/// TU2.cpp
static Monoid operator + (Monoid a, Monoid b) {
  return {
    a.value * b.value
  };
}

Monoid mult(Monoid a, Monoid b) {
  return a + b;
}

/// main.cpp
int main(int argc, char **argv) {
  auto v1 = sum({5}, {6}).value;
  auto v2 = mult({5}, {6}).value;
  std::cout << v1 << " " << v2 << "\n";
}
```

то оно даже будет работать ожидаемым образом, ведь никакой проблемы нет — определения локальны в единицах трансляции.

В C++17 дополнительными значениями обросло еще и ключевое слово `inline`.

Когда-то оно было лишь подсказкой компилятору о том, что тело функции надо "встраивать" вместо вызова, т. е. не делать относительно дорогой `call` с сохранением точки возврата, регистров, еще чего-то, а прямо вместо вызова воткнуть инструкции... Подсказка эта, правда, не всегда работает. По разным причинам. Но в основном потому, что программисты писали и пишут ее налево и направо даже туда, куда это делать не стоит, чтобы не раздувать чрезмерно получаемый код. Но это не наша история. Наша история о другом.

В современном C++ `inline` используется чаще всего только для того, чтобы поместить определение функции в заголовочный файл. В C это тоже работает, но совсем не так — вместо ошибки multiple definition, к которой приводит помещение не-`inline`-функций в заголовочный файл и которой мы хотели избежать, мы вовсе получили undefined reference.

В C `inline`-определения из заголовка нужно сопрячь модификатором `static`. И, возможно, получить code bloating, потому что вы получите копию функции в каждой единице трансляции, и все они будут считаться разными, если линковщик окажется недостаточно умным.

Либо все-таки предоставить одно не-`inline`-определение где-нибудь, например, вот таким мерзким трюком:

```
// square.h
#ifdef DEFINE_STUB
#define INLINE
```

```
#else
#define INLINE inline
#endif

INLINE int square(int num) {
  return num * num;
}

// square.c
#define DEFINE_STUB
#include "square.h"

// main.c
#include "square.h"

int main() {
  return square(5);
}
```

Или же упомянуть где-нибудь объявление этой функции со спецификатором `extern`[12]:

```
// square.h
inline int square(int num) {
  return num * num;
}

// square.c
#include "square.h"
extern int square(int num);

// main.c
#include "square.h"

int main() {
  return square(5);
}
```

Либо использовать GCC и собирать сишный код всегда с включенными оптимизациями. Только release-сборки! Я таких разработчиков тоже видел.

[12] Иногда работает и без него. `extern` сообщает, что определение находится в другом ~~замке~~ месте.

Но работает это решение не всегда:

```
// square.h
inline int square(int num) {
  return num * num;
}

inline int cube(int num) {
  return num * num * num;
}

// main.c
#include "square.h"
#include <stdlib.h>

typedef int (*fn) (int);

int main() {
  fn f;
  if (rand() % 2) {
    f = square;
  } else {
    f = cube;
  }
  // Адреса inline-функции неизвестны ->
  //   неопределенная ссылка
  return f(5);
}
```

Но вернемся к C++. Помимо функций, в заголовках иногда очень хочется определять еще и переменные. В приличных проектах, конечно, в основном константы. Но разработка сложна, темна и полна ужасов, а также нестандартных креативных решений, которые пришлось принять здесь и сейчас. Поэтому встречаются не только константы.

К сожалению, в C++ до 17-го стандарта просто так взять и поместить в заголовочный файл определение какой-то константы было не всегда возможно. А если и возможно, то с интересными спецэффектами.

```
// my_class.hpp
struct MyClass {
  static const int max_limit = 5000;
};

// main.cpp
#include "my_class.hpp"
```

```
#include <algorithm>

int main() {
  int limit = MyClass::max_limit; // OK
  return std::min(5, MyClass::max_limit); // Ошибка компиляции!
    // std::min хочет принять ссылку,
    // но линкер не знает адрес этой константы!
}
```

Можно написать:

```
// my_class.hpp
struct MyClass {
  static constexpr int max_limit = 5000;
};
```

И оно заработает.

Но constexpr возможен не всегда, и тогда все-таки придется взять и отнести определение в отдельную единицу трансляции...

Пришел C++17, и нашим мучениям настал конец! Теперь можно написать inline у переменной, и компилятор это съест, сгенерирует подобающую аннотацию для символа в объектном файле, чтобы линковщик более не кричал на multiple definition. Пусть берет любое — мы гарантируем, что все определения одинаковые, а иначе — undefined behavior.

```
// my_class.hpp
#include <unordered_map>
#include <string>

struct MyClass {
  static const inline
  std::unordered_map<std::string, int> supported_types_versions =
  {
    {"int", 5},
    {"string", 10}
  };
};

inline const
std::unordered_map<std::string, int> another_useful_map = {
  {"int", 5},
  {"string", 6}
};
```

```
void test();

// my_class.cpp
#include "my_class.hpp"
#include <iostream>

void test() {
  std::cout << another_useful_map.size() << "\n";
}

// main.cpp
#include "my_class.hpp"
#include <algorithm>
#include <iostream>

int main() {
  std::cout << MyClass::supported_types_versions.size() << "\n";
  test();
}
```

Всё прекрасно работает — никаких multiple definitions и никаких undefined references! Невероятно похорошел C++ при 17-м стандарте!

Внимательный читатель уже должен был почувствовать и даже заметить подвох.

Вот перед вами блок кода:

```
DEFINE_NAMESPACE(details)
{
  class Impl { ... };

  static int process(Impl);

  static inline const
    std::vector<std::string> type_list = { ... };
};
```

Может ли что-то пойти не так?

Конечно же, может! Это же C++!

`DEFINE_NAMESPACE(name)` может быть определен как:

```
#define DEFINE_NAMESPACE(name) namespace name
```

А может быть как:

```
#define DEFINE_NAMESPACE(name) struct name
```

Что?! Да! Что, если из благих побуждений, чтобы спрятать доступ к перегрузке функции `process` от вездесущего ADL, однажды сумрачному гению автора библиотеки пришло в голову именно такое решение, которое включается и выключается всего одним макросом!

В таких случаях вообще-то `type_list` — это другое.

В случае *`namespace`* — это `static inline` глобальная переменная. `inline` тут как бы бесполезен, потому что `static` модифицирует видимость глобальной переменной (linkage). В каждой единице трансляции, где такой заголовок окажется подключенным, будет своя копия переменной `type_list`.

В случае же `class` или `struct` этот `static inline` — поле, ассоциированное с классом, и оно будет одно на всех.

Ну ладно, какая разница! Они же константы и объявлены одинаково! Никто ничего не заметит на практике... Разумеется.

А теперь мы вспоминаем, что иногда нам нужны не константы. Например, если мы опять-таки делаем эту избитую систему с автоматической регистрацией плагинов при загрузке библиотек или иную систему авторегистрации типов.

Вот так всё работает. Красиво и ожидаемо.

```
// plugin_storage.h
#include <vector>
#include <string>
using PluginName = std::string;
struct PluginStorage {
  static inline std::vector<PluginName> registered_plugins;
};

// plugin.cpp
#include "plugin_storage.h"

namespace {
struct Registrator {
  Registrator() {
    PluginStorage::registered_plugins.push_back("plugin");
  }
} static registrator_;
}

// main.cpp
#include "plugin_storage.h"
#include <iostream>
int main() {
```

```
  // Печатает ровно один элемент.
  for (auto&& p : PluginStorage::registered_plugins) {
    std::cout << p << "\n";
  }
}
```

Меняем `struct PluginStorage` на `namespace PluginStorage` — всё компилируется, но уже не работает. Переменная `PluginStorage` своя в каждой единице трансляции, поэтому в `main` мы видим пустой список. Нужно удалить `static` перед `inline`, и мы получим желаемое поведение снова.

Поведем итог.

Изменяемые глобальные статические переменные — это сложно везде. В Rust, например, обращение к ним обязательно требует `unsafe`. C++ ничего не требует. Вам нужно помнить о множественных синтаксических ритуалах, которые нужно произвести:

- спрятать в функцию, чтобы избежать static initialization order fiasco;
- не написать лишних `static`;
- не запихнуть по неосторожности в заголовочный файл;
- максимально ограничить доступ.

И еще не забыть про многопоточный доступ.

C++17 породил переменные `static inline`. Они удобные, но только когда неизменяемые. Хотя и не беспроблемные. Средства просмотра изменений на ревью могут показывать не весь файл, а лишь часть с добавлением. Если видите `static inline`, не забудьте посмотреть, в каком он контексте. Если его проигнорировать, в лучшем случае ваши исполняемые файлы будут тяжелыми, в худшем — можно уйти во многие часы безнадежной отладки после какого-нибудь минималистичного изменения: кто-то объявление переменной с глобальным состоянием в заголовок вынес или, наоборот, внес, логически же ничего не поменялось...

Изменяемые статики — страшное зло. С ними не только у рядовых разработчиков проблемы. Например, для Clang более года висел баг, связанный с порядком инициализации статиков внутри одной единицы трансляции из-за неправильной сортировки `static` глобальных переменных и `static inline` полей классов.

Нарушение правила ODR

Вызвать функцию, которая не должна вызываться, испортить стек, сломать проверенную временем стороннюю библиотеку, довести до безумия программиста, пытающегося найти проблему под отладчиком, — всё может ODR (one definition rule) violation!

Вполне естественное и понятное правило, действующее во многих языках программирования: у одной и той же сущности должно быть не больше одного опре-

деления. Возьмем для примера функции. Их реализации могут различаться. Наличие двух и более определений функции приводит к проблеме: а какое же использовать?

В некоторых языках неопределенности нет. Например, в Python каждое следующее определение перекрывает предыдущее:

```
# hello.py
def hello():
    print("hello world")

hello() # hello world

def hello():
    print("Hello ODR!")

hello() # Hello ODR!
```

В иных языках[13] множественные определения просто приводят к ошибке компиляции.

```
fun x y = x + y

gun x y = x - y

fun x y = x * y

main = print $ "Hello, world!" ++ (show $ fun 5 6)

--    Multiple declarations of 'fun'
--    Declared at: 1124215805/source.hs:3:1
--                 1124215805/source.hs:7:1
```

C и C++ не исключения — в них переопределения функций, классов, шаблонов тоже диагностируются и выливаются в ошибку компиляции.

```
int fun() {
  return 5;
}

int fun() { // CE: повторное определение
  return 6;
}
```

И вроде бы всё хорошо. Ожидаемое, отличное решение. Но есть нюансы.

[13] Это Haskell.

Для статического анализа, конечно, очень удобно, если весь ваш код живет в одном-единственном файле. Но на практике обычно код разделяют на отдельные "модули", занимающиеся своей обособленной логикой. И вполне встречается ситуация, в которой два разных модуля содержат одноименные типы или функции. И это не должно вызывать проблем, должно работать из коробки... Но не в C и C++.

Знакомые с Python, наверное, знают, что в нем каждый отдельный файл — модуль — отдельное пространство имен. Имена классов и функции из разных файлов никак не интерферируют до тех пор, пока не будут импортированы.

В C никогда модулей не было и, скорее всего, не будет. Вместо них — раздельная компиляция, работающая на возможности оставлять сущности объявленными (например, в "подключаемых" заголовочных файлах), но не определенными (определение помещают в отдельную единицу трансляции, компилируемую независимо). Окончательная сборка и разрешение всех неопределенных имен откладываются до этапа линковки.

Никаких пространств имен также нет, и определение двух функций с одним и тем же именем в разных единицах трансляции нарушает правило ODR и... почти наверняка не будет отловлено на этапе компиляции. Возможно, если вам повезет и вы не забыли настроить опции линковки, проблема будет выявлена на следующем этапе. А если же вам не повезет, вы попадете в цепкие лапы неопределенного поведения.

Наибольшую неприятность доставляет то, что проблема не ограничивается сборкой лишь вашего кода. Ведь вы можете случайно использовать какое-то имя, встречающееся в сторонней библиотеке! И тогда можно сломать эту библиотеку как в своем проекте, так и в чужом, если ваш код будет использоваться в качестве зависимости. Причем достаточно случайно угадать лишь имя функции: в C нет перегрузок функции и определение функции с тем же именем, но с другими аргументами — ODR violation.

Из-за всех этих проблем в стандартах C и C++ даже указаны ограничения[14] на имена, которые вы можете использовать в своем коде, чтобы случайно не сломать стандартную библиотеку!

Что же делать?

В мире чистого C с этим борются комплексом методов:

1. Ручной имплементацией механизма пространств имен. Каждой функции и структуре в проекте дописывают префиксом имя проекта.
2. Настраивают видимость символов, добавляя:
   - `static`, который делает функцию или глобальную переменную "невидимой" за пределами единицы трансляции;
   - `__attribute__((visibility("hidden")))` для приватных структур и функций;

[14] Например, нигде нельзя использовать подряд два нижних подчеркивания. См. § 17.4.3.1.2 "Global names" в документации (**https://clck.ru/3JD567**).

- флаги `-fvisibility=hidden`, `-fvisibility-inlines-hidden` и выставление атрибутов только для публичного интерфейса.

3. Пишут скрипты для линкера, если предыдущий пункт пропустил в итоговый бинарь что-то лишнее.

Всё это, возможно, спасет при интеграции с другими библиотеками. Но от переопределения ваших функций и структур внутри вашего же проекта почти не помогает.

В C++ ситуация немного лучше.

Во-первых, есть перегрузки функций: типы аргументов участвуют в формировании имен, используемых при линковке, так что всего лишь угадать имя недостаточно, чтобы окунуться в неприятности, нужно еще угадать аргументы (но не тип возвращаемого значения!).

Во-вторых, есть пространства имен, и вручную прописывать префиксы к каждой объявляемой функции не нужно.

В-третьих, есть анонимные пространства имен, позволяющие делать невидимым за пределами единицы трансляции всё, что определено внутри него.

```
// A.cpp

namespace {
  struct S {
    S() {
      std::cout << "Hello A!\n";
    }
  };
}

void fun_A() {
  S{};
}

// B.cpp

namespace {
  struct S {
    S() {
      std::cout << "Hello B!\n";
    }
  };
}

void fun_B() {
  S{};
}
```

Структуры S находятся в разных анонимных пространствах имен, проблем с нарушением правила ODR не возникает.

У меня в проекте долгое время существовали два определения вспомогательной приватной структуры префиксного дерева, но не были помещены в анонимное пространство имен. Всё прекрасно работало до тех пор, пока однажды не поменяли порядок компиляции файлов. И сразу ошибка сегментации (segfault) — в объявлениях были разные типы полей, и при тестировании происходило настоящее безумие. Хорошо, что это обнаружилось раньше, чем упало на боевом стенде.

Наконец, в C++, начиная с 20-го стандарта, появились модули. Приватные, явно неэкспортируемые имена внутри одного модуля не интерферируют с именами из других модулей. Но для экспортируемых имен все проблемы сохраняются: объявлять пространство имен и следить за пересечениями надо самостоятельно.

Вместе с возможностями чуть реже нарушать ODR, в C++, конечно же, есть дополнительные возможности для неявного нарушения ODR — шаблоны.

Шаблоны инстанциируются в каждой единице трансляции и при использовании одних и тех же параметров должны раскрываться в один и тот же код, чтобы не нарушить правило ODR.

В C++ мы можем определять функции, принадлежащие к какому угодно пространству имен, в любой единице трансляции. А шаблоны компилируются в два прохода с привлечением ADL (argument dependent lookup). И горе вам, если один из проходов вытянет разные функции!

```
struct A {};
struct B{};
struct D : B {};

// demo_1.cpp
bool operator<(A, B) { std::cout << "demo_1\n"; return true; }
void demo_1() {
  A a; D d;
  std::less<void> comparator;
  comparator(a, d); // Шаблонный оператор ()
                    // ищет подходящее определение для <.
}

// demo_2.cpp
bool operator<(A, D) { std::cout << "demo_2\n"; return true; }
void demo_2() {
  A a; D d;
  std::less<void> comparator;
  comparator(a, d);
}
```

```
int main() {
  demo_1();
  demo_2();
  return 0;
}
```

В этом примере[15] разный порядок компиляции дает различные результаты:

- либо оба вызова печатают `demo_1`;
- либо оба вызова печатают `demo_2`;
- возможно даже поиграть с компилятором так, чтобы работало как задумывалось. Это остается читателю в качестве домашнего задания.

Занятно, что из-за специфики и трудности реализации двухэтапной компиляции шаблонов разные компиляторы будут давать различные результаты, если поместить этот пример в одну единицу трансляции! И о проблеме никто не сообщит!

Для упрощения анализа печать строк заменена печатью чисел 1 и 2.

GCC:

```
demo_1():
  mov   esi, 1
  mov   edi, OFFSET FLAT:_ZSt4cout
  jmp   std::basic_ostream<char, std::char_traits<char> >::operator<<(int)

demo_2():
  mov   esi, 1
  mov   edi, OFFSET FLAT:_ZSt4cout
  jmp   std::basic_ostream<char, std::char_traits<char> >::operator<<(int)
```

MSVC:

```
void demo_1(void) PROC                          ; demo_1, COMDAT
  push   2
  mov    ecx,
         OFFSET
         std::basic_ostream<char,std::char_traits<char> > std::cout
           ; std::cout
  call   std::basic_ostream<char,std::char_traits<char> > &
         std::basic_ostream<char,std::char_traits<char> >::operator<<(int)
           ; std::basic_ostream<char,std::char_traits<char> >::operator<<
  ret    0
void demo_1(void) ENDP

void demo_2(void) PROC                          ; demo_2, COMDAT
  push   2
```

[15] Пример прислал Дмитрий Лапшин (@LDVSOFT).

```
  mov    ecx,
         OFFSET
         std::basic_ostream<char,std::char_traits<char> > std::cout
           ; std::cout
  call   std::basic_ostream<char,std::char_traits<char> > &
         std::basic_ostream<char,std::char_traits<char> >::operator<<(int)
           ; std::basic_ostream<char,std::char_traits<char> >::operator<<
  ret    0
```

Код, собранный GCC, печатает `11`, MSVC — `22`.

Страшно? Не бойтесь! Если в этом примере `operator <` действительно предполагался приватным, то заворачивание его в анонимное пространство имен решило бы проблему. Внутри `std::less<void>::operator()` оператор `<` не был бы найден, и вы бы получили ошибку компиляции[16]. Пришлось бы использовать сравнение явно, и тут уже всё определено.

Используйте модули или помещайте детали реализации в анонимные пространства имен, и будет вам счастье. Наверное.

Если модули вам всё еще недоступны, в мире C и C++ существует особый подход к организации процесса сборки проекта, который способен отлавливать ODR violation — unity build. Но это, скорее, побочный эффект. Сам же unity build прежде всего предназначен для ускорения сборки за счет сокращения числа включений препроцессированных заголовков.

Например, если у вас есть два класса-компонента, разнесенных в независимые компилируемые файлы

```
// unit.cpp
#include <algorithm>
struct Unit {};
// weapon.cpp
#include <algorithm>
struct Weapon {};
```

то при независимой компиляции этих файлов каждый из них будет препроцессирован отдельно, а благодаря печально известному заголовку `<algorithm>` каждый препроцессированный файл выйдет величиной в десятки тысяч строк!

Например, с GCC 13.2:

```
g++ -E unit.cpp | wc -l
17710  # 17 тысяч строк!
g++ -E unit.cpp | wc -c
443562 # около 443 KB!
```

[16] Эта ошибка вам бы очень не понравилась, если ваш компилятор — GCC 11.4 или старее. 488 строк, полных template argument deduction/substitution failed. Более свежие версии компиляторов менее многословны.

Подход же unity build состоит в том, чтоб предварительно включить все компилируемые файлы в один общий:

```
// unity_build.cpp
#include "unit.cpp"
#include "weapon.cpp"
```

И разумеется, в таком случае, если мы нарушим правило ODR. Использовав в разных файлах одно и то же имя для различных сущностей, мы получим ошибку компиляции — ведь теперь у нас один файл компилируется!

Однако стоит иметь в виду, что у unity build есть серьезные недостатки.

- Этот подход может уменьшить время сборки с нуля, но значительно увеличить время пересборки при изменениях — нужно будет перекомпилировать всё!
- Вместо ODR violation можно внезапно получить выбор неправильных перегрузок и сопутствующие баги в логике!

```
// weapon.cpp
namespace {
double calculate_damage(double x) {
    return x * 10;
}
}
double Weapon::damage() {
    return calculate_damage(15.0);
}
// unit.cpp
namespace {
int calculate_damage(int x) {
    return x / 10;
}
}
void Unit::assign_damage(double x) {
    this->hp -= calculate_damage(x);
}
```

При включении файлов в порядке

```
#include "unit.cpp"
#include "weapon.cpp"
```

поведение метода `Unit::assing_damage` будет таким же, как при независимой сборке.

```
Unit::assign_damage(double):
        ...
        mov     edi, eax
        call    (anonymous namespace)::calculate_damage(int)
```

Но при включении в другом порядке

```
#include "weapon.cpp"
#include "unit.cpp"
```

перегрузка из анонимного пространства имен в файле *weapon.cpp* будет более подходящей, и поведение поменяется!

```
Unit::assign_damage(double):
        ...
        movq    xmm0, rax
        call    (anonymous namespace)::calculate_damage(double)
```

ODR violation почти всегда ходит рука об руку с проблемами обновлений и слома ABI.

Вы обновили библиотеку, и теперь ваш код зависит от ее более новой версии. Убедитесь, что другой код, зависящий от вашего, также использует новую версию этой библиотеки. Или хотя бы бинарно совместимую. Иначе — ODR violation, слом стека, нарушение конвенции вызова... ну, вы в курсе.

Слом ABI, потенциальное нарушение правила ODR — одни из самых острых причин, почему миграция на новые версии стандарта, компиляторов и библиотек в C++-мире занимает многие годы. Нужно всё пересобрать, всё перетестить, убедиться, что никто не привнес неправильных имен.

Как это ни парадоксально, но возможность нарушить ODR иногда оказывается полезной. Связанное с ним неопределенное поведение является в каком-то смысле определенным и контролируемым: какое именно из определений будет использоваться, задается порядком, на который можно влиять. GCC, например, поддерживает `__attribute__((weak))` для пометки функций, которые ожидаемо будут замещаться альтернативными определениями (с более эффективной реализацией, без отладочных инструкций, например). Или же техника symbol hooking, использующая `LD_PRELOAD`, чтобы заменить определенные функции из динамических библиотек для отладки с инструментированным аллокатором либо для перехвата вызовов и сбора статистики.

Зарезервированные имена

Эта тема тесно связана с нарушением правила ODR.

В C и C++ невероятно много идентификаторов, использовать которые для своих переменных и типов запрещено под страхом неопределенного поведения.

Некоторые имена запрещены самими стандартами C и C++, некоторые — стандартами POSIX, отдельные — платформоспецифическими библиотеками. В последнем случае вам обычно ничего не грозит, пока библиотека не подключена.

Так, в глобальной области видимости нельзя использовать имена функций из библиотеки C. Ни в C, ни в C++! Иначе вы можете столкнуться не только с ODR violation, но еще и с удивительным поведением компиляторов, умеющих оптимизировать распространенные конструкции.

Так, если определить собственный memset:

```
void *memset (void *destination, int c, unsigned long n) {
  for (unsigned long i = 0; i < n; ++i) {
    ((char*)(destination))[i] = c;
  }
  return destination;
}
```

заботливый оптимизирующий компилятор может запросто превратить его в:

```
void *memset (void* destination, int c, unsigned long n) {
  return memset(destination, c, n);
}
```

В C++, благодаря включенному по умолчанию декорированию имен, рекурсии не будет — вызовется стандартный memset вместо нашего.

Однако декорирование не спасает, если объявлять не функции, а глобальные переменные:

```
#include <iostream>
int read;
int main(){
  std::ios_base::sync_with_stdio(false);
  std::cin >> read;
}
```

При сборке такого примера со статически влинкованной стандартной библиотекой C программа упадет (SIGSEGV), т. к. вместо адреса стандартной функции read будет подставлен адрес глобальной переменной read. Аналогичный пример с использованием имени write предлагается читателю воплотить самостоятельно в качестве упражнения.

Запретных имен много. Например, всё, что начинается с is*, to* или _*, запрещено в глобальном пространстве. _[A-Z]* запрещены вообще везде. POSIX резервирует имена, заканчивающиеся на _t. И еще много всего неожиданного.

Кстати, эти пространства имен еще нельзя и расширять собственными сущностями.

Вы можете расширить пространства имен std или POSIX. Несмотря на то что такая программа успешно компилируется и исполняется, модификация этих пространств имен может привести к неопределенному поведению программы, если иное не указано стандартом.

Содержимое пространства имен std определяется исключительно Комитетом стандартизации, и стандарт запрещает добавлять в него:

- декларации переменных;
- декларации функций;
- декларации классов/структур/объединений;

- декларации перечислений;
- декларации шаблонов функций, классов и переменных (C++14).

Стандарт разрешает добавлять следующие специализации шаблонов, определенных в пространстве имен `std`, если они зависят хотя бы от одного определенного в программе типа (program-defined type):

- полная или частичная специализация шаблона класса;
- полная специализация шаблона функции (до C++20);
- полная или частичная специализация шаблона переменной, не лежащей в заголовочном файле '<type_traits>' (до C++20).

Однако специализации шаблонов, лежащих внутри классов или шаблонов классов, запрещены.

В отличие от пространства имен `std`, какая-либо модификация пространства имен POSIX полностью запрещена.

Если вы пользуетесь запрещенными именами, то сегодня может всё работать, но не завтра.

Чтобы не жить в страхе, во многих случаях достаточно использовать `static` или анонимные пространства имен. Или просто не использовать C и C++.

Если вы целенаправленно хотите заменить какую-нибудь стандартную функцию своей версией, обратите внимание на флаги `-fno-builtin`.

Нарушение правила ODR и разделяемые библиотеки

Ранее я рассматривал ODR violation в общих чертах и предупреждал о том, что может произойти, если случайно выбрать не то имя переменной, структуры или функции в C++. В этом же разделе я бы хотел продемонстрировать более изящный пример, не требующий приложения никаких усилий по написанию кривого кода. Достаточно просто иметь кривой код в ваших third-party-зависимостях.

Недавно я имел дело со странным баг-репортом:

Во внутреннем репозитории с пакетами обновился пакет с библиотекой GTest (известная уважаемая библиотека для написания самых разных тестов на C++). И в результате обновления некоторые тесты в конечных приложениях стали внезапно падать.

Падать они стали по-разному: у одних стали валиться проверяющие ассерты, у других же все работало, проверки проходили, но ctest рапортовал, что тестирующий процесс вышел с ненулевым кодом возврата.

Если с первыми происходило что-то совершенно невразумительное, то со вторыми можно было работать.

Запускаем тест вручную:

```
5/5 passed. Segmentation Fault.
```

О!

Запустив тест под отладчиком и выведя бэктрейс, я получил нечто следующего вида:

```
Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7e123fe in __GI___libc_free (mem=0x55555556a) at
./malloc/malloc.c:3368
3368    ./malloc/malloc.c: No such file or directory.
(gdb) bt
#0  0x00007ffff7e123fe in __GI___libc_free (mem=0x55555556a) at
./malloc/malloc.c:3368
#1  0x00007ffff7fb75ea in
std::vector<std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> >,
std::allocator<std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> > > >::
~vector() () from ./libgtest.so
#2  0x00007ffff7db2a56 in __cxa_finalize (d=0x7ffff7fb5090) at
./stdlib/cxa_finalize.c:83
#3  0x00007ffff7fb2367 in __do_global_dtors_aux () from
./libgmock.so
__do_global_dtors_aux () from ./libgmock.so
```

Что-то страшное и одновременно прекрасное произошло, осталось лишь понять, что именно.

Я прошелся по тестам: они были сгруппированы по отдельным исходникам, и каждый из них собирался в свой исполняемый файл. Все тесты собирались одними и теми же параметрами компиляции. Конфигурация задавалась в CMake попростому:

```
file(GLOB files "*_test.cpp")
foreach(file ${files})
  add_test(..., ${file})
endforeach()
```

Я открыл исходники падающего и непадающего тестов: падающий тест использовал `gMock`, а непадающий не использовал. Но при этом оба исполняемых файла были слинкованы с библиотекой *libgmock.so*.

Посмотрим еще раз на фрагмент бэктрейса:

```
std::allocator<char> > > >::~vector() () from ./libgtest.so
#2  0x00007ffff7db2a56 in __cxa_finalize (d=0x7ffff7fb5090) at
./stdlib/cxa_finalize.c:83
```

```
#3  0x00007ffff7fb2367 in __do_global_dtors_aux ()
from ./libgmock.so
```

Финализация глобальных объектов в `libgmock` как-то связана с деструктором глобальной переменной в `libgtest`.

Я открыл список изменений. Что же там такое обновилось во внутреннем пакете с GTest?..

```
Commit: ...
Produce shared libraries along with statics
```

И ровно две строчки, добавляющие в его *CMakeLists.txt* еще и динамические версии библиотек `libgmock` и `libgtest`.

Интересно. Я вышел в Интернет с этим вопросом и обнаружил что-то очень похожее[17]. Первым делом, глянув на дату issue, а также сверив даты коммитов во внутренней версии, я был глубоко разочарован темпами обновления зависимостей (ведь на дворе был конец 2023 года, а issue датируется 2016 годом). Но это уже совсем другая проблема C++...

Что же произошло на самом деле?

Фреймворк GoogleTest содержал две библиотеки:

- GTest — core-библиотека со всеми причиндалами для автоматической регистрации тестов и легкого их запуска через макрос `RUN_ALL_TESTS`;
- GMock — библиотека специально для mock-тестирования.

`gMock` линкуется с GTest. Конечный пользовательский исполняемый файл с тестами линкуется с обеими библиотеками.

Всё нормально, ничего криминального в этом нет.

Для поддержки всей красоты автоматической регистрации тестов и фикстур (вам, как пользователю, не нужно никуда складывать тестовые функции, вы их просто объявляете с помощью макросов) GTest очень активно полагается на глобальные переменные.

Проблемным объектом, чей деструктор приводил к падению, оказался, как видно из бэктрейса, вектор строк в *libgtest.so*. В исходниках GTest я обнаружил, что этот вектор — глобальная переменная, куда `InitGoogleTest()` складывает распознанные аргументы командной строки. Просто глобальная переменная, объявленная в компилируемом файле. Она не была в заголовочном файле. Всё вроде бы должно было быть хорошо... за одним исключением: она не была помечена `static` и не была обернута в анонимный `namespace`.

И что? Ведь всё же работало? Да, работало. Хитрость в том, как собирается библиотека GMock. Воспроизведем всё пошагово.

Заведем свой GTest дома.

```
// gtest.h
#pragma once
```

[17] См. **https://github.com/google/googletest/issues/930**.

```
void initGoogleTest(int argc, char* argv[]);
void runTests();

// gtest.cpp
#include "gtest.h"

#include <vector>
#include <string>
#include <iostream>

// Глобальная переменная,
// не static, как было в gtest
std::vector<std::string> g_args;

void runTests() {
  // Просто для демонстрации.
  std::cout << "run gtest\n";
  for (const auto& arg : g_args) {
    std::cout << arg << " ";
  }
  std::cout << "\n";
}

void initGoogleTest(int argc, char* argv[]) {
  for (int i = 0; i < argc; ++i) {
    g_args.push_back(argv[i]);
  }
}
```

Соберем его в статическую библиотеку, ведь именно статические библиотеки были изначально в пакете.

```
g++ -std=c++17 -fPIC -O2 -c gtest.cpp
ar rcs libgtest.a gtest.o
```

Добавим свой gMock.

```
// gmock.h
#pragma once
void runMocks();

// gmock.cpp
#include "gmock.h"
#include "gtest.h" // gmock линкуется с gtest!
```

```
#include <iostream>

void runMocks() {
  // Для демонстрации.
  std::cout << "run Mocks:\n";
  runTests();
}
```

Соберем его также в статическую библиотеку.

```
g++ -std=c++17 -fPIC -O2 -c gmock.cpp
ar rcs libgmock.a gmock.o gtest.o  #
```

И начнем пользоваться.

```
// main.cpp
#include "gtest.h"
#include "gmock.h"

int main(int argc, char* argv[]) {
  initGoogleTest(argc, argv);
  runMocks();
  runTests();
  return 0;
}

g++ -std=c++17 -O2 -o main main.cpp -L . -lgtest -lgmock
./main 1 2 3 4
run Mocks:
run gtest
./main 1 2 3 4
run gtest
./main 1 2 3 4
```

Должно быть очевидно, что, прилинковав обе библиотеки — GTest и GMock, мы уже как бы нарушили правило ODR: `gMock` содержит в себе GTest, а значит, у нас две копии глобальной переменной. Но всё работает: линкер выкинул одну из копий.

А теперь **добавим** динамические версии:

```
g++ -shared -fPIC -o libgtest.so gtest.o
g++ -shared -fPIC -o libgmock.so gtest.o gmock.o
```

и пересоберем клиентский код:

```
g++ -std=c++17 -O2 -o main main.cpp -L . -lgtest -lgmock
LD_LIBRARY_PATH=. ./main
run Mocks:
run gtest
```

```
./main
run gtest
./main
Segmentation fault (core dumped)
```

Ура! Падает! Обе библиотеки имеют собственную версию глобальной переменной с одним и тем же неявно экспортируемым именем. Использоваться опять-таки будет только **одна**. После загрузки библиотеки, после конструирования глобальной переменной стандарт C++ требует зарегистрировать — например, через `__cxa_atexit` — функцию для вызова деструктора. У нас две библиотеки, значит, две функции будут вызваны. На одном и том же объекте. Double free. Конструктор, кстати, также вызывается дважды по одному и тому же адресу:

```
struct GArgs : std::vector<std::string> {
  GArgs() {
    std::cout << "Construct it: " <<
              uintptr_t(this) << "\n";
  }
};

GArgs g_args;

LD_LIBRARY_PATH=.  ./main 1 2 3
Construct it: 140368928546992
Construct it: 140368928546992
run Mocks:
run gtest
./main 1 2 3
run gtest
./main 1 2 3
Segmentation fault (core dumped)
```

И это прекрасно. Ладно, теперь проблема ясна. Тесты, которые падали на ассертах, также страдали, но от других глобальных переменных. Осталась последняя проблема: я упомянул, что все тесты собирались одинаково, но какие-то не падали — те, что не использовали `gMock`, но все равно с ним линковались.

Закомментируем использование нашего `gMock`:

```
#include "gtest.h"
#include "gmock.h"

int main(int argc, char* argv[]) {
  initGoogleTest(argc, argv);
  // runMocks();
  runTests();
```

```
  return 0;
}

g++ -std=c++17 -O2 -o main main.cpp -L . -lgtest -lgmock
LD_LIBRARY_PATH=.  ./main 1 2 3
Construct it: 140699707998384
run gtest
./main 1 2 3
```

О как! Конструктор вызвался только раз. Значит, и деструктор будет вызван только раз. Всё отлично. Но ведь мы же линковали... Современные линковщики достаточно умны, чтобы не тащить то, что не используется (что, кстати, иногда является проблемой, если у конструкторов в библиотеке есть побочные эффекты).

Если мы заставим GCC прилинковать `gMock` насильно:

```
g++ -std=c++17 -O2 -o main main.cpp
    -Wl,--no-as-needed -L . -lgtest -lgmock
LD_LIBRARY_PATH=.  ./main 1 2 3
Construct it: 139687276064944
Construct it: 139687276064944
run gtest
./main 1 2 3
Segmentation fault (core dumped)
```

всё будет сломано, как и должно.

Подобный паттерн по созданию проблем оказывается невероятно распространенным! И не только в C++.

`LibA` статически влинковывается в `LibB` и обе они (`LibA` и `LibB`) влинковываются в `BinC`. Самый частый кандидат на такую `LibA` — библиотеки менеджмента памяти.

Например, при сборке динамических библиотек в Rust и подключении их в другие Rust-проекты практически всегда люди натыкаются на эту проблему: `std` статически влинковывается и в библиотеку, и в исполняемый файл.

Я также обнаружил подобные проблемы в AWS SDK: глобальный `UniquePtr` также двумя путями попадает в конечное приложение, потому в его деструктор воткнули зануление, чтобы не вызывать `delete` дважды.

Тривиальные типы и ABI

Допустим, вы написали прекрасную библиотеку для работы с двумерными векторами. И там, конечно же, была структура `Point`. Вот такая:

```
template <typename T>
struct Point {
  ~Point() {}
```

```
  T x;
  T y;
};
```

И была функция

```
Point<float> zero_point();
```

имплементацию которой вы, как приличный разработчик, заботящийся о времени компиляции и размерах заголовочных файлов, поместили в компилируемый *.cpp*-файл, а пользователю оставили только объявление. Ваша библиотека была такой хорошей, что быстро обрела популярность, и от нее стали зависеть многие другие приложения.

И всё было хорошо. Но однажды вы заметили, что деструктор `Point` вам совершенно не нужен. Пусть его генерит компилятор самостоятельно. И вы его удалили.

```
template <typename T>
struct Point {
  T x;
  T y;
};
```

Пересобрали библиотеку и разослали пользователям новые готовые *.dll/.so*-файлы. С новыми заголовками, конечно же. Но изменение такое незначительное, что пользователи просто подложили готовые бинари себе без перекомпиляции... и всё упало со страшным memory corruption.

Почему?

Это изменение сломало ABI.

В C++ все типы делятся на тривиальные и нетривиальные. Тривиальные, в свою очередь, бывают еще и в разных аспектах тривиальными[18]. В общем случае тривиальность позволяет не генерировать дополнительный код, чтобы что-то сделать.

- `trivially_constructible` — не надо ничего инициализировать.
- `trivially_destructible` — не нужно генерировать код для деструктора.
- `trivially_copyable` — не нужно ничего, кроме простого копирования байтов.
- `trivially_movable` — аналогично копированию, но при перемещении.

С объектами тривиальных типов выполнимы дополнительные оптимизации. Например, их можно передавать через регистры. Компилятор способен догадаться соптимизировать `memcpy` (использованный, чтобы избежать неопределенного поведения) в `reinterpret_cast` и выполнить другие подобные действия.

Вот, например:

```
struct TCopyable {
  int x;
  int y;
```

[18] Из-за того что тривиальность бывает разных категорий, термин *trivial type* и предикат `std::is_trivial` признаны и помечены устаревшими в C++26.

```
};
static_assert(std::is_trivially_copyable_v<TCopyable>);

struct TNCopyable {
  int x;
  int y;

  TNCopyable(const TNCopyable& other) :
    x{other.x}, y{other.y} {}

  // Вынуждены написать конструктор, так как
  // aggregate initialization отключился из-за
  // конструктора копирования.
  TNCopyable(int x, int y) : x{x}, y{y} {}
};

static_assert(!std::is_trivially_copyable_v<TNCopyable>);

// Здесь будет возврат через регистр rax.
// TCopyable в него как раз помещается.
extern TCopyable test_tcopy(const TCopyable& c) {
  return {c.x * 5, c.y * 6};
}

// Здесь возврат через указатель,
// передаваемый через регистр rdi.
extern TNCopyable test_tnocopy(const TNCopyable& c) {
  return {c.x * 5, c.y * 6};
}
```

По ассемблерному листингу можно убедиться, что две "одинаковые" функции, возвращающие "одинаково" представленные в памяти структуры, делают это по-разному:

```
test_tcopy(TCopyable const&):     # @test_tcopy(TCopyable const&)
  mov     eax, dword ptr [rdi]
  mov     ecx, dword ptr [rdi + 4]
  lea     eax, [rax + 4*rax]
  add     ecx, ecx
  lea     ecx, [rcx + 2*rcx]
  shl     rcx, 32
  or      rax, rcx  #!
  ret
```

```
test_tnocopy(TNCopyable const&):  # @test_tnocopy(TNCopyable const&)
  mov     rax, rdi
  mov     ecx, dword ptr [rsi]
  mov     edx, dword ptr [rsi + 4]
  lea     ecx, [rcx + 4*rcx]
  add     edx, edx
  lea     edx, [rdx + 2*rdx]
  mov     dword ptr [rdi], ecx     #!
  mov     dword ptr [rdi + 4], edx #!
  ret
```

Аналогично и с вашей 2D-точкой:

```
struct TPoint {
  float x;
  float y;
};
static_assert(std::is_trivially_destructible_v<TPoint>);

struct TNPoint {
  float x;
  float y;
  ~TNPoint() {} // Деструктор, предоставляемый пользователем,
                // делает тип нетривиальным,
                // даже если деструктор ничего не делает.
};

static_assert(
  !std::is_trivially_destructible_v<TNPoint>);

// Возврат через регистр.
extern TPoint zero_point() {
  return {0,0};
}

// Возврат через указатель.
extern TNPoint zero_npoint() {
  return {0,0};
}

zero_point():                    # @zero_point()
  xorps   xmm0, xmm0
  ret
```

```
zero_npoint():                  # @zero_npoint()
  mov     rax, rdi
  mov     qword ptr [rdi], 0
  ret
```

Особенно болезненно всё становится с шаблонами, поскольку тривиальность шаблонной структуры может зависеть от тривиальности параметров.

Возьмем, например, реализацию шаблона `pair` в стандартной библиотеке C++03. В нем можно увидеть `user-provided`, ничего дополнительно не делающие, конструкторы копий. В C++11 и более позних версиях их уже нет. Так что это еще одна точка бинарной несовместимости библиотек на старом C++ с библиотеками нового C++.

Проблемы со сломом ABI вокруг тривиальных типов могут застать вас врасплох не только в самом C++, но и в любом другом языке, если вы попытаетесь через него общаться с плюсовыми библиотеками. Например, в утилите для генерации биндингов для Rust есть баг[19].

Будьте внимательны с тривиальными типами. И если вы намерены предоставлять стабильный ABI своей библиотеки, то выстраивайте его вокруг чистых сишных структур и функций, а не вокруг зоопарка из мира C++.

Также помните, что тривиальность:

- легко ломается (достаточно добавить инициализатор, и тип уже не `trivially_constructible`);
- соседствует с неинициализированными полями, а чтение неинициализированных переменных заканчивается неопределенным поведением;
- не всегда влияет на ABI (в основном влияют деструкторы и конструкторы копирования/перемещения).

Неинициализированные переменные

Это очень известный и распространенный источник проблем не только в C или C++.

Новые современные языки программирования обычно запрещают использование неинициализированных переменных. Либо переменные всегда инициализируются значением по умолчанию (например, в Go), либо попытка чтения из неинициализированной переменной дает ошибку компиляции (в Kotlin или в Rust).

C и C++ — старые языки. В них можно легко и просто объявить переменную, а инициализировать ее когда-нибудь потом. Или забыть инициализировать вовсе. Но в отличие от совсем низкоуровневого ассемблера, в котором читать из неинициализированной переменной никто не запрещает (ну получите вы свои мусорные байтики, и ладно), в C и C++[20] это влечет за собой неопределенное поведение.

---

[19] Для нетривиальных типов bindgen генерирует функции с несовместимым ABI (см. **https://github.com/rust-lang/rust-bindgen/issues/778**).

[20] А также в Rust, см. тип `MaybeUninit` (**https://doc.rust-lang.org/std/mem/union.MaybeUninit.html**).

Но время не стоит на месте даже для C++. В последних версиях стандарта (C++26 и новее) всё же произошли некоторые изменения: стандарт вводит новое понятие ошибочного (erroneous) поведения. И ошибка чтения неинициализированной переменной считается теперь ошибочным, а не неопределенным поведением. На практике же это значит, что вы все также успешно отстрелите себе ногу, но компиляторам рекомендуется выдать диагностику и запрещается делать оптимизации — какой-то определенный мусор должен быть успешно прочитан, а оптимизации кода до и после такого чтения не должны делать никаких предположений об этом мусорном значении.

Например,

```
void test() {
    int x;
    if (x) {
        printf("non zero\n");
    } else {
        printf("zero\n");
    }
}
int main() {
    test();
    test();
}
```

Сейчас Clang 19 с ключом `-O3` оптимизирует код выше в ничто, считая код недостижимым из-за неопределенного поведения в нем!

```
test():

main:

ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 48
```

В C++26 ожидается, что все-таки какой-то результат должен быть.

При этом если прочитанный мусор представит собой все-таки невалидное значение для данного типа переменной, то все спецэффекты неопределенного поведения остаются в силе!

Неожиданный вариант такого UB можно наблюдать на следующем примере[21]:

```
struct FStruct {
  bool uninitializedBool;
```

[21] Пример взят из вопроса на StackOverflow: "Does the C++ standard allow for an uninitialized bool to crash a program?" (см. **https://clck.ru/3JDTdf**).

```
  // Конструктор, не инициализирующий поля.
  // Чтобы проблема воспроизвелась,
  // конструктор должен быть определен
  // в другой единице трансляции.
  // Можно сымитировать с помощью атрибута noinline.
  __attribute__ ((noinline)) FStruct() {};
};

char destBuffer[16];

void Serialize(bool boolValue) {
  const char* whichString = boolValue ? "true" : "false";
  size_t len = strlen(whichString);
  memcpy(destBuffer, whichString, len);
}

int main()
{
  // Конструируем объект с неинициализированным полем.
  FStruct structInstance;

  // Неопределенное поведение!
  Serialize(structInstance.uninitializedBool);

  //printf("%s", destBuffer);
  return 0;
}
```

Программа падает. Поскольку неинициализированных переменных в корректной программе не бывает, компилятор полагает `boolValue` всегда валидным и выполняет следующую занятную оптимизацию:

```
// size_t len = strlen(whichString); // 4 или 5!
   size_t len = 5 - boolValue;
```

"Физическое" значение неинициализированной bool-переменной — это необязательно `true`/`false`, а, например, `7`. Поэтому в некоторых случаях получится `len = 5 - 7`. Как следствие — переполнение буфера при вызове `memcpy`.

Так если отсутствие неинициализированных переменных способствует оптимизациям, почему бы их не запретить совсем с жесткой ошибкой компиляции?

Во-первых, они позволяют экономить на спичках:

```
int answer;
if (value == 5) {
  answer = 42;
```

```
} else {
  answer = value * 10;
}
```

Если бы нам было запрещено объявлять переменную без инициализации, мы бы вынуждены были либо написать

```
int answer = 0;
```

и потратить в отладочной сборке целую одну лишнюю инструкцию `xor` на зануление!

Либо завернуть вычисление `answer` в отдельную функцию (или лямбда-функцию) и получить целый `call` вместо `jmp`, если компилятор не проведет оптимизацию!

Либо использовать тернарный оператор и получить что-то совершенно нечитаемое, если веток условий будет больше.

Во-вторых, иногда спички большие и дорогие, и экономия оправдана:

```
constexpr int data_size = 4096;
char buffer[data_size];
read(fd, buffer, data_size);
```

Инициализировать целый массив, чтобы тут же его перетереть, не разумно. И маловероятно, что компилятор эту инициализацию выбросит при оптимизации: для этого ему нужны гарантии, что условная функция `read` не читает ничего из буфера. Такие гарантии могут быть зашиты для функций стандартной библиотеки, но не для пользовательских.

Как избежать неинициализированные переменные

Сначала разберемся: какие конструкции порождают неинициализированные переменные?

Специальные функции, например `std::make_unique_for_overwrite`, мы не рассматриваем. Как и функции выделения сырой памяти (`*alloc` тоже). Хочу напомнить, что написать `(T*)malloc(N)` в ожидании инициализированной памяти нельзя. Для этого есть `calloc`.

В более общем случае, если верно, что `is_trivially_constructible<T> == true`, то

```
T x;
T x[N];
T* p = new T;
T* p = new T[N];
```

порождают неинициализированные переменные/массивы (или указатели на неинициализированные переменные/массивы).

Если тип нетривиально конструируемый, не спешите радоваться. Его конструктор по умолчанию мог забыть что-то проинициализировать. Или тип просто потерял тривиальность только потому, что кто-то предоставил деструктор, чтобы вас запутать, или объявил виртуальный метод, а инициализацию так и не обеспечил.

Нетривиально конструируемый тип

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

struct S {
  int uninit;
  ~S() {}
};

static_assert(!std::is_trivially_constructible_v<S>);

int main()
{
  // Здесь в стеке локально конструируем экземпляр нашей структуры.
  // Булев член uninitializedBool остается неинициализированным.
  S uninit1;
  std::cout << uninit1.uninit << "\n";
  S uninit2[2];
  std::cout << uninit2[1].uninit << "\n";
  S* puninit = new S;
  std::cout << puninit->uninit << "\n";
  S* puninit2 = new S[2];
  std::cout << puninit2[1].uninit << "\n";

  int* p = new int;

  std::cout << *p << "\n";
  // Печатаем в стандартном выводе "true" или "false".
  return 0;
}
```

Этот код компилируется с помощью Clang 18.1, несмотря на ключи `-Wall -Wextra -O3 -std=c++17 -Wpedantic -Wuninitialized`, и выводит мусор:

```
272846040
-658903036
-658903036
-658903036
-658903036
```

С GCC 14.1 и теми же ключами этот код выводит нули, а компилятор выдает предупреждения, но не во всех случаях.

```
<source>: In function 'int main()':
<source>:17:36: warning: 'uninit1.S::uninit' is used uninitialized
[-Wuninitialized]
   17 |     std::cout << uninit1.uninit << "\n";
      |                                    ^~~~
<source>:16:7: note: 'uninit1.S::uninit' was declared here
   16 |     S uninit1;
      |       ^~~~~~
<source>:19:39: warning: 'uninit2[1].S::uninit' is used
uninitialized [-Wuninitialized]
   19 |     std::cout << uninit2[1].uninit << "\n";
      |                                      ^~~~
<source>:18:7: note: 'uninit2' declared here
   18 |     S uninit2[2];
```

Но стоит лишь убрать ключ `-O3`, и предупреждения исчезнут!

Распространенный совет по повсеместному использованию {} при объявлении переменных работает и гарантирует инициализацию нулями только с тривиальными типами. Для нетривиальных — всё на совести конструктора.

Но иногда вам может "повезти", и инициализация пройдет (гарантированно стандартом!) в два этапа: сначала нулями, потом вызовется конструктор по умолчанию. Мне удалось воспроизвести этот эффект только при использовании `std::make_unique;`.

Как бороться с неинициализированными переменными и связанным с ними неопределенным поведением?

- Не разрывать объявление и инициализацию. Вместо этого использовать конструкции:

  ```
  auto x = T{...};
  auto x = [&] { ... return value }();
  ```

- Проверять свои конструкторы, чтобы в них были инициализированы все поля.
- Пользоваться инициализаторами по умолчанию при объявлении полей структур.
- Использовать свежие версии компиляторов: они все лучше учатся выявлять обращения к неинициализированным значениям.
- Не использовать `new T`, если вы не уверены в том, что делаете. Всегда `new T{}` или `new T()`.
- Не забывать про динамический и статический анализ внешними утилитами. Valgrind умеет ловить обращения к неинициализированной памяти. В PVS-Studio этой проблеме посвящен целый пучок диагностик.

И последнее

Если вам когда-нибудь придет светлая мысль использовать неинициализированную память в качестве источника случайности, гоните ее (мысль) как можно быстрее! Некоторые пробовали — не получилось[22].

Бесконечные диапазоны в C++20

Поддержка работы как с кастомными, так и со стандартными коллекциями в C++ от версии к версии все лучше и лучше.

Алгоритмы стандартной библиотеки образца 11–17-х стандартов работали и работают с парами итераторов, задающих диапазон элементов коллекции.

```
const std::vector<int> v = {1,2,3,4,5};
std::vector<int> odds;
std::copy_if(
  v.begin(), v.end(), std::back_inserter(odds),
  [](int x){ return x % 2 == 0;});
std::vector<int> squares;
std::transform(
  odds.begin(), odds.end(), std::back_iserter(squares),
  [](int x) { return x * x;});
// Возвращаем квадраты.
```

Многословно, неудобно. Да еще и совсем не zero cost — лишние аллокации обычно ни один компилятор C++ не оптимизирует. Но, конечно, мы можем провести все оптимизации самостоятельно — алгоритмы старого STL, работающие с итераторами, довольно гибкие в выборе того, что и как вы хотите сделать.

```
std::vector<int> v = {1,2,3,4,5};
v.erase(
  std::remove_if(v.begin(), v.end(),
                 [](int x){ return x % 2 != 0; }),
  v.end()
);
std::transform(v.begin(), v.end(), v.begin(),
               [](int x){return x * x;});
return v;
```

Отлично, ни одной лишней аллокации! Но всё так же многословно и путанно. Да еще и странно выглядящая конструкция `erase-remove`.

[22] OpenSSL использовал так неинициализированные переменные... Результат: CVE-2008-0166.

Большинству людей обычно нужно сначала написать простое и понятное решение, а потом уже его оптимизировать по мере надобности. Простыми и понятными решения, использующие старые алгоритмы над парами итераторов, назвать сложно.

В C++11 появился range-based for, и стало удобно просто итерироваться по коллекции.

```
for (auto x : v) {
  // что-то проделываем с x
}
```

Но так итерироваться можно лишь по всей коллекции. А что, если вы хотите от пятого элемента в векторе до десятого? Пиши́те цикл со счетчиком либо используйте std::for_each с парой итераторов.

```
std::for_each(v.begin() + 5, v.begin() + 10, [&](auto x) {
  // что-то делаем
});
```

Либо вам нужно откуда-то — из книг, курсов или из самого стандарта — узнать, что range-based-for автоматически работает для любого объекта, у которого есть методы begin и end.

begin() и end() должны возвращать итераторы. Если они возвращают что-то другое, то в 99,9% случаев вы получите ошибку компиляции[23]. В экзотических случаях может быть что-то неожиданное.

Из всего этого возникает вполне здравая идея: а что, если для итерирования по части коллекции сделать структуру с итераторами? И для всяких transform, filter, reverse...[24]

И вот мы уже разрабатываем свою библиотеку для удобной работы с коллекциями, также удобно работающую с range-based-for. И всё хорошо до тех пор, пока нас не посещает идея: а не сделать ли ленивую процедурно генерируемую последовательность с совместимым интерфейсом?

Например, мы можем сделать "бесконечный" генератор чисел:

```
struct Numbers {
  struct End {};  // Тип-маркер для проверки, достигли ли конца.
  struct Number { // Сам итератор-генератор. Требуются 3 операции.
    int x;
    // Проверка достижения конца.
    bool operator != (End) const {
      return true;
    }
```

[23] Иногда вразумительную. В GCC 14.2, например: error: inconsistent begin/end types in range-based 'for' statement.

[24] Ух! Да это же как раз C++20 ranges.

```
    // Генерация текущего элемента.
    int operator*() const {
      return x;
    }
    // Продвижение итератора - переход к следующему элементу.
    Number operator++() {
      ++x;
      return *this;
    }
  };

  explicit Numbers(int start) : begin_{start} {}

  Number begin_;

  auto begin() { return begin_; }
  End end() { return {}; }
};
```

И вот тут начинается засада.

Ни старые алгоритмы STL, ни `range-based-for` не работают — не компилируются, потому что требуют, чтобы `begin` и `end` имели одинаковый тип.

Хорошо, мы можем исправить это относительно безболезненно в нашем простеньком примере:

```
struct Numbers {
  struct Number {
    int x;
    bool operator != (Number) const {
      return true;
    }
    int operator*() const {
      return x;
    }
    Number operator++() {
      ++x;
      return *this;
    }
  };

  explicit Numbers(int start) : begin_{start} {}

  Number begin_;
```

```
  auto begin() { return begin_; }
  auto end() { return begin_; }
};
```

Правда, семантика `operator !=` стала странной. Да и нужно `end()` из чего-то конструировать. Если состояние нашего генератора будет более сложным, например выделяющим что-то на куче, мы получим дополнительные накладные расходы. Не очень zero-cost.

Поэтому в C++17 `range-based-for` исправили. Теперь он может работать с граничными итераторами разных типов.

Но STL-алгоритмы всё так же не работают.

```
auto nums = Numbers(10);

// Ошибка компиляции:
auto pos = std::find_if(nums.begin(), nums.end(),
                        [](int x){ return x % 7 == 0;});

std::cout << *pos;
// Ошибка компиляции...
```

В C++20 наконец-то всё пофиксили. Нет, старые STL-алгоритмы всё так же не работают. Просто теперь есть новые STL-алгоритмы: почти такие же, как старые, только в пространстве имен `std::ranges` и жестко требующие удовлетворения новых концептов итераторов. Поэтому пример ниже слегка распухает.

```
struct Numbers {
  struct End {

  };
  struct Number {
    using difference_type = std::ptrdiff_t;
    using value_type = int;
    using pointer = void;
    using reference = value_type;
    using iterator_category = std::input_iterator_tag;

    int x;
    bool operator == (End) const {
      return false;
    }

    int operator*() const {
      return x;
    }
```

```
    Number& operator++() {
      ++x;
      return *this;
    }
    Number operator++(int) {
      auto ret = *this;
      ++x;
      return ret;
    }
  };

  explicit Numbers(int start) : begin_{start} {}

  Number begin_;

  auto begin() { return begin_; }
  End end() { return {}; }
};
```

С ними следующий код компилируется и работает.

```
auto nums = Numbers(10);

auto pos =
  std::ranges::find_if(nums.begin(), nums.end(),
                       [](int x){ return x % 7 == 0;});
std::cout << *pos;
```

Что ж. Это было небольшое введение. Теперь мы, наконец, можем начать отстреливать себе ноги.

Нам поможет std::unreachable_sentinel

Выдумывать на каждый итератор, генерирующий бесконечную последовательность, новый тип (`EndSentinel`) для метода `end()` нам благодаря C++20 не надо. В стандартной библиотеке определен тип `std::unreachable_sentinel_t`, задизайненный именно для этой цели. Он сравним на равенство с любым объектом, "похожим" на `ForwardIterator`. И результат сравнения всегда отрицательный.

С ним наш пример с числами упрощается.

```
struct Numbers {
  struct Number {
    using difference_type = std::ptrdiff_t;
    using value_type = int;
    using pointer = void;
```

```
    using reference = value_type;
    using iterator_category = std::input_iterator_tag;

    int x;
    int operator*() const {
      return x;
    }
    Number& operator++() {
      ++x;
      return *this;
    }
    Number operator++(int) {
      auto ret = *this;
      ++x;
      return ret;
    }

  };

  explicit Numbers(int start) : begin_{start} {}

  Number begin_;

  auto begin() { return begin_; }
  auto end() { return std::unreachable_sentinel; } // !
};
```

Сравнение с `unreachable_sentinel` не требует выполнения никаких операций. Так что его можно использовать, например, чтобы сформировать `range`, итерирование по которому будет происходить без проверки границ.

Например:

```
// Если у нас есть вектор, задающий перестановку
std::vector<size_t> perm = { 1, 2, 3, 4, 5, 6, 7, 8, 9};
std::ranges::shuffle(perm, std::mt19937(std::random_device()()));

// и нам поступают запросы на поиск позиции элемента,
// заведомо находящегося в векторе
// size_t p = 7;
assert(p < perm.size());
return std::ranges::find(perm.begin(),
                         std::unreachable_sentinel, p) - perm.begin();
```

Очевидно, это крайне небезопасный ход, к которому стоит прибегать только в случае, если вы точно всё проверили, и эта оптимизация критична и необходима. Если в примере выше по какой-то причине будет запрошен элемент, не присутствующий в векторе, мы получим неопределенное поведение.

Рефакторинг больших участков кода, использующего подобные фичи, может закончиться поиском трудноуловимых багов. В отличие от Rust, в C++ мы не можем гарантированно пометить участок кода как потенциально опасный и проблемный. В C++ любой участок кода потенциально небезопасен, и подчеркнуть это можно только комментарием или какими-нибудь ухищрениями в именовании функций или переменных.

Невиртуальные виртуальные функции

Вы разрабатываете иерархию классов и хотите описать интерфейс для вычислителя, который можно запускать и останавливать. Скорее всего, в первой итерации он будет выглядеть так:

```
class Processor {
public:
  virtual ~Processor() = default;
  virtual void start() = 0;
  // Прекращает выполнение. Если уже остановлен, возвращает 'false'.
  virtual bool stop() = 0;
};
```

Пользователи интерфейса нареализовывали своих имплементаций. Все были счастливы, пока кто-то не сделал асинхронную реализацию. С ней почему-то приложение стало падать. Проведя небольшое расследование, вы выяснили, что пользователи интерфейса не позаботились вызвать метод `stop()` перед разрушением объекта. Какая досада!

Вы были уставшими и злыми. А, быть может, это были и не вы, а какой-то менее опытный коллега, которому поручили доработать интерфейс. В общем, на свет родилась правка:

```
class Processor {
public:
  virtual void start() = 0;
  // Прекращает выполнение. Если уже остановлен, возвращает 'false'.
  virtual bool stop() = 0;
  virtual ~Processor() {
    stop();
  }
};
```

Логично? Да!

Правильно? Нет!

Если вам повезет, то анализатор кода или достаточно умный компилятор сможет сообщить о проблеме.

```
GCC 11.2: warning: pure virtual 'virtual bool Processor::stop()' called from
destructor
```

В конструкторах и деструкторах в C++ виртуальная диспетчеризация методов не работает. В других языках, например в C# или Java, наоборот, доставляет свои проблемы.

Почему так? При конструировании часть объекта-наследника, используемая в переопределенном методе, может быть еще не создана: конструкторы вызываются в порядке от базового класса к производному. При деструктурировании, наоборот, часть объекта-наследника уже уничтожена, и если позволить динамический вызов, можно легко получить ситуацию use-after-free.

Радуйтесь! Это одно из немногих мест в C++, где вас защитили от неопределенного поведения, связанного с временем жизни!

Хорошо. А если так? Добавим промежуточный вызов функции.

```
// processor.hpp
class Processor {
public:
  void start();
  // Прекращает выполнение. Если уже остановлен, возвращает 'false'.
  bool stop();

  virtual ~Processor();

protected:
  virtual bool stop_impl()  = 0;
  virtual void start_impl() = 0;
};



// processor.cpp
Processor::~Processor() {
  stop();
}

bool Processor::stop() {
  return stop_impl();
}
```

```
void Processor::start() {
  start_impl();
}
```

Компиляторы уже не выдают замечательного предупреждения, и подвох заметить стало сложнее. Анализатор PVS-Studio так просто не обмануть, и пока он продолжает выдавать

```
V1053 Calling the 'stop_impl' virtual function indirectly in the destructor may lead
to unexpected result at runtime. Check lines: 18, 22, 11.
```

А ведь мы повысили уровень индирекции всего на один! А что будет, если код нашего базового класса окажется сложнее?.. Наследование имплементаций — источник многих проблем, прячущихся за невинным желанием использовать код повторно.

Вызов виртуальных функций класса в его конструкторах и деструкторах почти всегда является ошибкой сейчас или в будущем. Если же это не ошибка, а так и задумывалось, то стоит использовать явный статический вызов с указанием имени класса (name qualified call).

```
// processor.cpp
Processor::~Processor() {
  Processor::stop();
}
```

Также стоит отметить, что в C++ у `pure virtual`-методов может быть имплементация, к которой можно обращаться. Иногда это даже полезно. Таким образом можно потребовать от пользователя обязательно принять решение: изменять поведение метода или использовать поведение по умолчанию.

```
class Processor {
public:
  virtual void start() = 0;
  // Прекращает выполнение. Если уже остановлен, возвращает 'false'.
  virtual bool stop() = 0;

  virtual ~Processor() = default;
};

void Processor::start() {
  std::cout << "unsupported";
}

class MyProcessor : public Processor {
public:
  void start() override {
```

```
    // Вызываем реализацию, заданную по умолчанию.
    Processor::start();
  }
};
```

Вернемся опять к нашей остановке при вызове деструктора. Как же с ней быть?

Есть два пути.

Путь первый: потребовать, чтобы реализующий интерфейс обязательно предоставил свою версию деструктора, которая выполнит корректную остановку.

Насильно, с проверкой на этапе компиляции, к этому, к сожалению, никого не принудишь. Можно попытаться выразить намерение объявлением деструктора интерфейса чисто виртуальным, но это не поможет, поскольку деструктор, если не указан, всегда генерируется.

```
class Processor {
public:
  virtual void start() = 0;
  // Прекращает выполнение. Если уже остановлен, возвращает 'false'.
  virtual bool stop() = 0;
  virtual ~Processor() = 0;
};

// Обязательно!
Processor::~Processor() = default;

class MyProcessor : public Processor {
public:
  void start() override {
  }
  bool stop() override { return false; }
  // Отсутствующий деструктор не провоцирует CE.
  // ~MyProcessor() override = default;
};

int main() {
  MyProcessor p;
}
```

Путь второй — добавить еще один слой и пользоваться им во всех публичных API.

```
class GuardedProcessor {
  std::unique_ptr<Processor> proc;
  // ...
  ~GuardedProcessor() {
```

```
    assert(proc != nullptr);
    proc->stop();
  }
};
```

Вишенка на торте: играя с деинициализацией объектов, легко получить дублирование операций освобождения ресурсов.

Освобождение ресурсов в деструкторах с использованием вспомогательных функций

Нужно было создать иерархию классов, управляющих некими ресурсами. Какой конкретно класс будет создан во время исполнения программы, заранее неизвестно, так что вооружаемся полиморфизмом, делаем деструктор как полагается виртуальным и будем вызывать методы только через указатели на базовый класс.

Начнем с простейшего базового класса.

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

class Resource
{
public:
  void Create() {}
  void Destroy() {}
};

class A
{
  std::unique_ptr<Resource> m_a;

public:
  void InitA()
  {
    m_a = std::make_unique<Resource>();
    m_a->Create();
  }

  virtual ~A()
  {
    std::cout << "~A()" << std::endl;
    if (m_a != nullptr)
      m_a->Destroy();
  }
};
```

```
int main()
{
  std::unique_ptr<A> p = std::make_unique<A>();
}
```

Пока всё хорошо. Запускаем:

```
~A()
```

Далее требования меняются! Иногда очень нужно сбрасывать состояние объекта, т. е. освобождать ресурсы, не дожидаясь вызова деструктора. Хорошо, интерфейс базового класса пополняется удобным виртуальным методом.

```
class A
{
  std::unique_ptr<Resource> m_a;

public:
  void InitA()
  {
    m_a = std::make_unique<Resource>();
    m_a->Create();
  }

  virtual void Reset()
  {
    std::cout << "A::Reset()" << std::endl;
    if (m_a != nullptr)
    {
      m_a->Destroy();
      m_a.reset();
    }
  }

  virtual ~A()
  {
    std::cout << "~A()" << std::endl;
    Reset();
  }
};

int main()
{
  std::unique_ptr<A> p = std::make_unique<A>();
}
```

Запускаем и получаем:

```
~A()
A::Reset()
```

Добавился виртуальный метод `Reset`, освобождающий ресурсы. Чтобы не дублировать код, деструктор теперь не сам освобождает ресурсы, а просто вызывает этот метод.

Мы уже видели ранее, что вызывать виртуальные методы в деструкторе не лучшая идея. Но пока всё хорошо. Да и метод же не чисто виртуальный. Никаких проблем пока нет...

Добавляем новый класс-наследник. У него будут собственные ресурсы вдобавок к существующим в базовом классе, и их тоже нужно не забыть освободить в методе `Reset`.

```
class A
{
  std::unique_ptr<Resource> m_a;

public:
  void InitA()
  {
    m_a = std::make_unique<Resource>();
    m_a->Create();
  }

  virtual void Reset()
  {
    std::cout << "A::Reset()" << std::endl;
    if (m_a != nullptr)
    {
      m_a->Destroy();
      m_a.reset();
    }
  }

  virtual ~A()
  {
    std::cout << "~A()" << std::endl;
    Reset();
  }
};
```

```
class B : public A
{
  std::unique_ptr<Resource> m_b;

public:
  void InitB()
  {
    m_b = std::make_unique<Resource>();
    m_b->Create();
  }

  void Reset() override
  {
    std::cout << "B::Reset()" << std::endl;
    if (m_b != nullptr)
    {
      m_b->Destroy();
      m_b.reset();
    }
    A::Reset();
  }

  ~B() override
  {
    std::cout << "~B()" << std::endl;
    Reset();
  }
};

int main()
{
  std::unique_ptr<A> p = std::make_unique<B>();
  p->Reset();
  std::cout << "------------" << std::endl;
  p->InitA();
}
```

Получаем

```
B::Reset()
A::Reset()
------------
```

```
~B()
B::Reset()
A::Reset()
~A()
A::Reset()
```

Если из внешнего кода явно вызываем метод `Reset`, всё отрабатывает как и задумывалось: вызывается `B::Reset()`, которая затем вызывает одноименный метод из базового класса.

А вот в деструкторах происходит нечто неожиданное. Каждый деструктор использует `Reset` своего класса, что порождает избыточный вызов.

Если мы будем и дальше наследоваться, продолжая переопределять деструкторы и метод `Reset`, то всё больше будем усугублять проблему. Всё больше и больше лишних вызовов.

Добавим еще один класс:

```
C::Reset()
B::Reset()
A::Reset()
-----------
~C()
C::Reset()
B::Reset()
A::Reset()
~B()
B::Reset()
A::Reset()
~A()
A::Reset()
```

Ошибка проектирования класса очевидна. Впрочем, это сейчас она очевидна нам. В реальном проекте такие ошибки вполне себе могут жить, оставаясь незамеченными: код ведь работает, и функции сброса состояния класса справляются со своей задачей. Проблема в избыточных действиях и неэффективности.

Заметив описанную ошибку и попытавшись ее исправить, программист рискует допустить две другие типовые ошибки.

Первый вариант. Объявить методы `Reset` невиртуальными и не вызывать в них базовые варианты (`x::Reset`). Тогда каждый деструктор будет вызывать только функцию `Reset` из своего класса и освобождать лишь свои ресурсы. Это действительно уберет повторные вызовы в деструкторах, однако поломает очистку состояния объекта при вызове `Reset` извне. Сломанный код распечатает:

```
A::Reset()   // Сломали очистку ресурсов извне.
-----------
~C()
C::Reset()
```

```
~B()
B::Reset()
~A()
A::Reset()
```

Второй вариант. Вызвать виртуальную функцию `Reset` однократно только в деструкторе базового класса. Это приведет к утечкам, т. к. согласно правилам C++ будет вызвана функция `Reset`, реализованная в базовом классе, а не в наследниках. Это логично, поскольку к моменту вызова деструктора `~A()` все наследники разрушены, и вызывать их методы нельзя. Сломанный код распечатает:

```
C::Reset()
B::Reset()
A::Reset()
------------
~C()
~B()
~A()
A::Reset()  // Утечка! Освобождены только ресурсы в базовом классе.
```

Данный пример навеян реальным случаем бага, найденным с помощью PVS-Studio в движке qdEngine. Так что это реальные ловушки. Будьте бдительны, проектируя подобные классы.

Как же правильно реализовать классы, чтобы избежать множественных избыточных вызовов?

Нужно отделить очистку ресурсов в классах от интерфейса для их очистки извне.

Лучше всего возложить ответственность за освобождение ресурсов на соответствующие деструкторы полей наших классов и не писать собственные деструкторы `~A()`, `~B()` и `~C()`. Но не всегда такое возможно: типы для полей могут быть предоставлены библиотекой, которую вы не можете поправить, а на создание дополнительных оберток у вас нет времени.

Следует создать невиртуальные функции, отвечающие только за очистку данных в классах, где они объявлены. Назовем их `ResetImpl` и сделаем приватными, т. к. они не предназначены для внешнего использования.

Тогда деструкторы смогут просто делегировать свою работу функциям `ResetImpl`.

Функция `Reset` останется публичной и виртуальной. Она будет очищать данные всех классов, используя всё те же вспомогательные функции `ResetImpl`.

Соберем все вместе и напишем корректный код:

```
class A
{
  std::unique_ptr<Resource> m_a;

  void ResetImpl()
  {
```

```
    std::cout << "A::ResetImpl()" << std::endl;
    if (m_a != nullptr)
    {
      m_a->Destroy();
      m_a.reset();
    }
  }

public:
  void InitA()
  {
    m_a = std::make_unique<Resource>();
    m_a->Create();
  }

  virtual void Reset()
  {
    std::cout << "A::Reset()" << std::endl;
    ResetImpl();
  }

  virtual ~A()
  {
    std::cout << "~A()" << std::endl;
    ResetImpl();
  }
};

class B : public A
{
  std::unique_ptr<Resource> m_b;

  void ResetImpl()
  {
    std::cout << "B::ResetImpl()" << std::endl;
    if (m_b != nullptr)
    {
      m_b->Destroy();
      m_b.reset();
    }
  }
```

```
public:
  void InitB()
  {
    m_b = std::make_unique<Resource>();
    m_b->Create();
  }

  void Reset() override
  {
    std::cout << "B::Reset()" << std::endl;
    ResetImpl();
    A::Reset();
  }

  ~B() override
  {
    std::cout << "~B()" << std::endl;
    ResetImpl();
  }
};

class C : public B
{
  std::unique_ptr<Resource> m_c;

  void ResetImpl()
  {
    std::cout << "C::ResetImpl()" << std::endl;
    if (m_c != nullptr)
    {
      m_c->Destroy();
      m_c.reset();
    }
  }

public:
  void InitC()
  {
    m_c = std::make_unique<Resource>();
    m_c->Create();
  }
```

```
  void Reset() override
  {
    std::cout << "C::Reset()" << std::endl;
    ResetImpl();
    B::Reset();
  }

  ~C() override
  {
    std::cout << "~C()" << std::endl;
    ResetImpl();
  }
};

int main()
{
  std::unique_ptr<A> p = std::make_unique<C>();
  p->Reset();
  std::cout << "------------" << std::endl;
}
```

И получим наконец-то желаемое:

```
C::Reset()
C::ResetImpl()
B::Reset()
B::ResetImpl()
A::Reset()
A::ResetImpl()
------------
~C()
C::ResetImpl()
~B()
B::ResetImpl()
~A()
A::ResetImpl()
```

Не сказать, что красиво получилось. Избыточность побеждена многословностью!

Массивы переменной длины

Память в C и C++ под наши объекты, как известно, можно выделять на стеке, а можно в куче. На стеке она обычно выделяется автоматически, и нам об этом сильно беспокоиться не надо.

```
int32_t foo(int32_t x, int32_t y) {
  int32_t z = x + y;
  return z;
}
```

При вызове функции `foo` на стеке после аргументов (хотя не факт, что аргументы будут переданы через стек) будет выделено (просто вершина стека будет сдвинута) еще 4 байта под переменную `z`. А может быть, и больше, кто ж знает, что там настроено у компилятора! А может быть, и не будет выделено. Например, если компилятор оптимизирует переменную и сложит результат сразу в регистр `rax`. Дикая природа удивительна, не правда ли?

Освобождается память со стека тоже автоматически, причем всегда. Если только, конечно, вы случайно не сломали стек, не сделали чудовищную ассемблерную вставку, и теперь адрес возврата не ведет куда-то не туда, или не используете `attribute ( ( naked ) )`. Но, мне кажется, в этих случаях у вас куда более серьезные проблемы... Во всех остальных случаях память со стека освобождается автоматически. Потому, как известно, вот такой код порождает висячий указатель:

```
int32_t* foo(int32_t x, int32_t y) {
  int32_t z = x + y;
  return &z;
}
```

Обычно для выделения чего-то на стеке размер этого чего-то должен быть заранее известен на этапе компиляции. Обычно, но не всегда...

Однажды один большой и сложный HTTP-сервер внезапно упал. Упал он, как ни странно, с моим любимым сообщением "segmentation fault (core dumped)". К этому все, впрочем, уже привыкли, ведь HTTP-сервер был написан на чистом и прекрасном C. Так что падение — это что-то само собой разумеющееся.

Содержимое core-файла было загадочным: строчка, на которую указывал dump, не делала ничего страшного. Она не разыменовывала указатель, не писала в массив, не читала из массива, не освобождала память, не выделяла память... Ничего. Она просто пыталась вызвать функцию и передать в нее параметры. Но что-то пошло не так.

Закончился стек.

Но как же так?! В core dump было всего от силы 40 стек-фреймов! Как он мог закончится? Там же 10 Мбайт под Linux!

Путешествуя по этому стек-трейсу, я поднялся на пять стек-фреймов выше. Все они были довольно небольшого размера: 40 байт, 180 байт, килобайт... А вот шестой фрейм оказался невероятно большим! 8 Мбайт!

Открыв соответствующий исходник, я обнаружил:

```
int encoded_len = request->content_len * 4 / 3 + 1;
char encoded_buffer[encoded_len];
```

```
encoded_len =
  encode_base64(request->content, content_len,
                encoded_buffer, encoded_len);
process(encoded_buffer, enconded_len);
```

Знакомьтесь, массив переменной длины (variable-length array, VLA)! Прекрасная фича языка C. Существует в языке C++ как нестандартное расширение (MSVC не поддерживает, в GCC и Clang компилируется).

Это массив переменной длины **на стеке**. Причина падения была ясна.

VLA — концептуально, фича довольно полезная. Но крайне небезопасная. Вам нужен буфер, но его длину вы узнаёте только во время выполнения? Пожалуйста, VLA! Не нужен никакой `malloc`/`new`? Просто объяви массив и укажи длину! К тому же это в среднем намного быстрее, чем `malloc`, и не утечет, автоматически освободится! Ну и, конечно же, что может быть лучше, чем получить segfault вместо out-of-memory?

Лучше VLA может быть только прямое использование функции `alloca()`. Ведь, в отличие от VLA, у нее намного больше вариативности для минирования вашего кода.

```
void fill(char* ptr, int n) {
  for (int i =0;i<n;++i) {
    ptr[i] = i * i;
  }
}

int use_alloca(int n) {
  char* ptr = (char*)alloca(n);
  fill(ptr, n);
  return ptr[n-1];
}

int main() {
  int n = 0;
  for (int i = 1; i < 10000; ++i) {
    n += use_alloca(i);
  }
  return n ? 0 : 1;
}
```

Тут каждый вызов `alloca` не приводит к переполнению стека сам по себе. Но если `use_alloca` будет заинлайнена компилятором по какой-либо причине, мы получим SIGSEGV.

Дополнительная прелесть в том, что вызов функции `alloca` может быть куда-то спрятан, и вы даже можете не догадываться, что ее используете. Например, `alloca` живет внутри макроса для конвертации строк A2W (библиотека ATL). Казалось бы,

вы просто конвертируете в цикле строчки. И всё даже работает, пока строчек мало и они короткие. А потом раз, и Stack Overflow. Если не знать такой нюанс устройства макроса, ее практически нереально найти, например, на code review. Ладно, именно про `alloca` в цикле PVS-Studio предупредит. Но в целом очень коварно.

Использование `alloca` и VLA крайне не рекомендуется. Некоторые стандарты кодирования (например, MISRA C) вообще запрещают использование VLA.

man[25] упоминает случай, когда их использование может быть оправдано: ваш код полагается на `setjmp`/`longjump`, и нормальный менеджмент динамически выделенной памяти может быть осложнен, а стек всё равно будет очищен даже при `longjmp`. Не буду спрашивать, зачем оно вам...

`alloca` и VLA, и правда, в среднем быстрее, чем динамическое выделение памяти (`new`, `malloc`). Но если уж нужно действительно быстро, то вариант с преаллоцированным массивом или массивом фиксированной длины получше будет.

Немного замеров

Код:

```
#include <vector>
#include <alloca.h>

int calc(char* arr, int n) {
    for (int i = 0; i < n; ++i) {
        arr[i] = (i * i) & 0xFF;
    }
    return arr[n-1];
}

int test_vla(int n) {
    char arr[n];
    return calc(arr, n);
}

int test_dyn(int n) {
    std::vector<char> v(n);
    return calc(v.data(), n);
}

int __attribute__ ((noinline)) test_alloca(int n) {
    char* arr = (char*)alloca(n);
    return calc(arr, n);
}
```

[25] man 3 alloca (см. **https://man7.org/linux/man-pages/man3/alloca.3.html**).

```
int test_fixed(int n) {
    char arr[10005];
    return calc(arr, n);
}

static void VLA(benchmark::State& state) {
  int size = 10000;
  for (auto _ : state) {
    if  (size < 10005) {
      size += 1;
    }
    benchmark::DoNotOptimize(test_vla(size));
  }
}
// Зарегистрируем функцию как бенчмарк.
BENCHMARK(VLA);

static void ALLOCA_TEST(benchmark::State& state) {
  int size = 10000;
  for (auto _ : state) {
    if  (size < 10005) {
      size += 1;
    }
    benchmark::DoNotOptimize(test_alloca(size));
  }
}
BENCHMARK(ALLOCA_TEST);

static void DYN(benchmark::State& state) {
  int size = 10000;
  for (auto _ : state) {
    if  (size < 10005) {
      size += 1;
    }
     benchmark::DoNotOptimize(test_dyn(size));
  }
}
BENCHMARK(DYN);

static void FIXED(benchmark::State& state) {
  // Код внутри этого цикла раз за разом измеряется.
  int size = 1;
  char fixed[10005];
```

```
  for (auto _ : state) {
    if  (size < 10005) {
      size += 1;
    }
    benchmark::DoNotOptimize(calc(fixed, size));
  }
}
BENCHMARK(FIXED);

static void FIXED_LOCAL(benchmark::State& state) {
  // Код внутри этого цикла раз за разом измеряется.
  int size = 1;
  for (auto _ : state) {
    if  (size < 10005) {
      size += 1;
    }
    benchmark::DoNotOptimize(test_fixed(size));
  }
}
BENCHMARK(FIXED_LOCAL);
```

А теперь графики.

А в C++ (без расширений) VLA нет. Там есть шаблоны, а они не дружат с VLA.

```
#include <iostream>

template <size_t N>
void test_array(int (&arr)[N]) {
  std::cout << sizeof(arr) << "\n";
}

int main(int argc, char* argv[]) {
  int fixed[15];
  int vla[argc];
  test_array(fixed);
  test_array(vla);      // Ошибка компиляции!
}
```

Владение, исключения и ошибки

Команде однажды завели баг-репорт: "Сервис упал с segmentation fault. В core dump стек-трейс указывает как последнюю функцию перед падением чего-то из вашей библиотеки. Разберитесь!" Упал сервис ровно один раз за полгода.

Этим "чего-то" был вызов `free` где-то глубоко-глубоко внутри библиотеки Protobuf. И несколько последующих стек-фреймов указывали на вызов деструктора уже в нашей библиотеке. Потратив некоторое время на анализ кода деструктора, дежурный инженер не нашел ничего подозрительного и предположил, что это похоже на какую-то ранее встреченную проблему в Protobuf. И как воспроизвести, никто не представлял. Тупик...

Я заинтересовался этой загадочной историей и залез в core dump поглубже.

На пару десятков стек-фреймов выше, уже принадлежащих чужому сервису, засветилась функция `lru_insert`. Уже интересно. Это оказалась функция вставки в LRU-кеш. И можно было заподозрить, что, возможно, вызов деструктора как-то связан с вытеснением объекта из кеша.

Я решил найти конкретный код того сервиса и понять, что же там происходит при вставке. Обнаруженный код привел меня сначала в замешательство, а потом в восхищение:

```
auto metadata = new Metadata(...);
metadata->cached = true;
lru_insert(cache, key, metadata);
// Проверяем, успешно ли прошла вставка!
if (auto item_handle = lru_get(cache, key)) {
...
} else {
// Не найдено -> не кешируется.
```

```
metadata->cached = false;
}
```

Если есть голый new, то где-то должен быть и delete... И я нашел его. Целых два.

Один при создании кеша:

```
auto cache =
lru_create(n, [](void* data){ delete static_cast<Metadata*>(data); });
```

А второй где-то в другом месте:

```
if (!metadata->cached) {
    delete metadata;
}
```

Дело пахнет повторным удалением. Но что же здесь пошло не так?

```
lru_insert(cache, key, metadata);
// Проверяем, успешно ли прошла вставка!
if (auto item_handle = lru_get(cache, key)) {
...
} else {
// Не найдено -> не кешируется.
metadata->cached = false;
}
```

Код восхитителен тем, что явно выполняет аж два обращения к кешу: на вставку и на проверку. Но ведь можно ограничиться только одной вставкой: если lru_insert предоставит информацию об успехе. Может ли дело быть в этом? Нет ли случайно в этом сервисе гонок, которые могут вклиниться между вставкой и проверкой? Но меня уверили, что процесс однопоточный.

Наверное, стоит углубиться в функцию lru_insert. Ее написали 10 лет назад и больше не трогали. Ее протестировали, она надежна. Как я могу в ней сомневаться?

```
void lru_insert(Cache* c, const char* key, void* data)
{
  try {
    c->cache.insert(std::string(key),
    boost::intrusive_pointer(new LRUItem(data,
    c->deleter)));
  } catch (...) {}
}
```

От увиденного мне стало дурно. Ведь здесь опять голый new, который мог[26] привести к утечке в самом редком и очень часто игнорируемом случае: если конструктор

[26] C++17 добавил более строгие гарантии на порядок вычисления аргументов, так что в вычислении boost::intrusive_pointer(new LRUItem...) ничего вклиниться не может. Тем не менее порядок вычисления аргументов на одном уровне все равно остается неспецифицированным. А знаменитое i++ + ++i — всё также не определено.

std::string бросит исключение (bad_alloc). И, как мы видим, эта функция полностью игнорирует пойманные исключения.

Но, как утверждает Rust, утечка — это совершенно безопасно[27]! И к ситуации use-after-free или double-free привести не может. А мы получили ошибку сегментации (segfault) с очень большой вероятностью именно из-за double-free.

Есть, конечно, еще вариант, что cache.insert может бросить исключение, не выполнить вставку и заставить intrusive_pointer удалить объект. Но этот сценарий не согласуется с увиденным core dump: повторное удаление (и падение) какого-то старого объекта произошло внутри insert. Если бы новый объект был удален, падение произошло бы в другом месте... Или бы не произошло вовсе. Неопределенное поведение!

У нас есть еще один подозреваемый в этом фрагменте кода. Давайте-ка глянем функцию lru_get. Ее тоже написали 10 лет назад, протестировали, и нет ни малейшего повода в ней сомневаться!

```
// LRUItemHandle защищает данные от удаления,
// если они покинули кеш в результате его опорожнения
LRUItemHandle* lru_get(Cache* c, const char* key) {
  try {
    auto item_ptr = c->cache.get(std::string(key));
    if (!item_ptr) {
      return nullptr;
    }
    return new LRUItemHandle(item_ptr);
  } catch (...) {
    return nullptr;
  }
}
```

Я был в ужасе. Надеюсь, вы тоже. Вернемся обратно к злосчастному фрагменту.

```
lru_insert(cache, key, metadata);
// Допустим, что insert отработал успешно.
if (auto item_handle = lru_get(cache, key)) {
  ...
} else {
  // У lru_get есть как минимум две возможности соврать нам
  // о наличии элемента в кеше. И похоже, что нас обманули.
  metadata->cached = false;
}
```

[27] В Rust есть методы и функции для потенциального создания утечек. Например, std::mem::forget. И они не требуют unsafe-блоков для вызова.

Как выяснилось, в экстремально редком случае, раз в полгода, в зависимости от нагрузки у сервиса заканчивалась память. Но вместо падения по out-of-memory он пытался продолжить работать. Таковы требования. И часто ему это удавалось делать успешно. Пока однажды `bad_alloc` не был выброшен в этой надежной и протестированной библиотеке с LRU-кешем.

Теперь, когда все улики собраны и преступление реконструировано, надо сделать шаг назад и обдумать произошедшее.

- Разработчики сервиса проиграли в игру с ручным разделением владения данными между разными компонентами и динамическим определением, кто эти данные будет освобождать. Это сложная игра.
- У библиотеки с LRU-кешем, с которым и хотели разделить владение, оказалось чудовищное API. Почему это C-API — обоснование тому есть, и оно не касается нашей истории. Но это чудовищное C-API.

```
// Эта функция не сообщает о возможной ошибке вставки.
// Эта функция пытается завладеть data, но в случае ошибки
// data может быть удалена или оставлена - зависит от типа ошибки:
// - не удалена при ошибке аллокации до входа в c->cache.insert
// - удалена при любой ошибке внутри c->cache.insert
void lru_insert(Cache* c, const char* key, void* data)

// Эта функция может упасть с ошибкой либо не найти элемент.
// Но различить эти два исхода пользователю
// не предлагается.
LRUItemHandle* lru_get(Cache* c, const char* key)
```

Передача и разделение владения через границы C-API с помощью сырых указателей и с учетом ошибок и исключений — непростая задача на внимательность, которая может стать кошмаром, если писать в стиле C, игнорируя возможности C++.

API библиотеки с "надежным и протестированным" LRU-кешем я расширил новыми функциями, более устойчивыми к ошибкам. И постарался исправить старые, насколько это возможно: например, проблему с владением в `lru_insert` нельзя было исправлять полностью... потому что нашелся пользователь, который полагался на неправильное поведение.

А функцию стоило написать так с самого начала:

```
// Эта функция принимает владение data и передает его кешу.
// В случае любой ошибки data НЕ будет освобождена.
// В случае успеха контролировать время жизни data будет кеш.
ErrorCode lru_try_insert(Cache* c, const char* key, void* data) try {
  // Подготавливаем слот для сырого указателя.
  auto slot =
    boost::intrusive_pointer(new LRUItem(nullptr, c->deleter));
```

```
  // Вставляем пустой слот в кеш.
  // Слот должен быть пуст, чтоб не удалить данные при ошибке вставки.
  c->cache.insert(std::string(key), slot);
  // Передаем владение в слот.
  // На этом этапе никакой больше ошибки произойти не может.
  // Деструктор LRUItem гарантирует вызов deleter(data)
  slot->data = data
  return ErrorCode::LRU_OK;
} catch (...) {
  return ErrorCode::LRU_ERROR;
}
```

Но лучше бы, конечно, пересмотреть зависимости зависимостей и использовать C++-библиотеку для LRU-кеша с более безопасными RAII-типами.

Исключения (или паники, как в Rust) всегда осложняют ручное управление ресурсами. Это специфично не только для низкоуровневых языков. Например, программы на Go, Java, C#, Python и других также страдают от незакрытых файлов или соединений с базами данных, если программист забыл использовать `try-with-resources`-, `finally`- или `defer`-блоки.

Ручное управление ресурсами крайне рекомендуется сводить к минимуму с помощью:

- RAII-оберток;
- умных указателей;
- умных аллокаторов и arena/region based-управления множеством объектов;
- любого другого способа, который больше подходит вашей задаче и вашим ресурсам.

Также крайне рекомендую C++-разработчикам пробовать Rust как минимум в качестве тьюториала по владению и его передаче и разделению. Строгий borrow checker раскроет вам много интересных паттернов, в которых при полностью ручном контроле можно легко получить double-free.

ГЛАВА 6
Происхождение указателей

Невалидные указатели

Что вообще такое указатель? Когда это понятие пытаются объяснить новичкам в C++, часто говорят, что это число, адрес, указывающий на номер ячейки в памяти, где что-то лежит.

Это в каком-то смысле справедливо на очень низком уровне — в ассемблере, в машинных кодах. Но в C и C++ указатель — это не просто адрес. И тем более не число, которое как-то просто по-особому используется. Более того, в C++ (не в C) есть указатели, которые вообще не являются адресами в памяти — указатели на поля и методы классов. Но о них мы говорить сейчас не будем.

Указатель — это ссылочный тип данных. Нечто, с помощью чего можно получить доступ к другим объектам. И, в отличие от ссылок C++, объекты-указатели являются настоящими объектами, а не странными псевдонимами для существующих значений. С числами и адресами в памяти указатели связаны только деталями реализации.

Для указателей в стандарте C++ подробно расписано, откуда они могут появляться. Если коротко, то:

- как результат применения операции взятия адреса (`&x` или `std::addressof(x)`);
- как результат вызова оператора `new` (возможно, `placement new`);
- как результат неявного преобразования имени массива или имени функции в указатель;
- как результат некоторой допустимой операции над другим указателем;
- как копирование существующего указателя, в частности, копирование `nullptr`.

Все остальные источники указателей определены реализацией (implementation defined) или вообще не определены (undefined).

Главная операция, выполняемая над указателями, — это разыменование, т. е. получение доступа к объекту, на который этот указатель ссылается. И вместе с этой операцией приходит главная проблема — ее не ко всем указателям применять можно. Существуют и другие операции, которые также применимы не к любому указателю. Но, конечно, есть единственная операция, допустимая почти всегда, — сравнение на равенство (неравенство).

В идеальном светлом мире от типа объекта зависит множество допустимых над ним операций. Но в случае указателей (и это очень печально) применимость или неприменимость зависит не только от значения указателя, но еще и от того, откуда этот указатель взялся, а также откуда взялись другие указатели!

```
int x = 5;
auto x_ptr  = &x; // Валидный указатель, его МОЖНО разыменовывать.

auto x_end_ptr  = (&x) + 1;    // Валидный указатель,
                               // но его НЕЛЬЗЯ разыменовывать.

auto x_invalid_ptr = (&x) + 2; // Невалидный указатель,
                               // само его существование недопустимо.
```

Сравнение указателей с помощью операторов "больше/меньше" определено только для указателей на элементы одного и того же массива. Для произвольных указателей — не уточнено (unspecified).

Арифметика указателей допустима лишь в пределах одного и того же массива (от указателя на первый элемент до указателя на элемент за последним). Иначе — undefined behavior. Особым является только случай `(&x) + 1` — любой объект считается массивом из одного элемента.

Пример кода, который валится с UB именно на арифметике указателей, найти сложно, зато можно привести пример с итераторами (которые разворачиваются в указатели).

```
std::string str = "hell";
str.erase(str.begin() + 4 + 1 - 3);
```

Этот код упадет в отладочной сборке под MSVC. `str.begin() + 4` — указатель на элемент за последним. И еще `+1` выводит за пределы строки. Это UB. И неважно, что дальше вычитание возвращает внутренний указатель обратно в границы строки.

Не стоит выполнять сложные вычисления с указателями. Прибавлять к ним или вычитать лучше всегда конечный числовой результат. В конкретном примере расчет отступа `(4 + 1 - 3)` нужно выполнить отдельно — расставить скобки. Еще лучше, безопаснее и понятнее — вынести в отдельную переменную.

Помимо выхода за границы объектов, невалидные указатели могут появляться после отрабатывания некоторых функций. Наиболее яркий пример такого UB представил Ник Левицки (Nick Lewycky) для Undefined Behavior Consequences Contest[1]. Немного переделанная под C++ версия (чтобы в ней было только одно UB, а не два) выглядит так:

```
int* p = (int*)malloc(sizeof(int));
int* q = (int*)realloc(p, sizeof(int));
if (p == q) {
```

[1] См. **https://blog.regehr.org/archives/767**.

```
  new(p) int (1);
  new(q) int (2);
  std::cout << *p << *q << "\n"; // выводим 12
}
```

Этот код, собранный Clang 18.1.0 (`-O3 -std=c++20`), выводит результат `12`, противоречащий здравому смыслу (если вы не знаете, что в коде UB!). И этот же пример демонстрирует, что указатели — это не просто число-адрес.

Указателем, переданным в сишную функцию `realloc`, при успешной реаллокации пользоваться более нельзя. Его можно только перезаписать (а потом уже использовать).

Этот пример, конечно, искусственный, но в него можно легко влететь, если, например, по какой-то причине писать свою версию вектора, используя `realloc`, и захотеть немного "соптимизировать".

```
template <class T>
struct Vector {
  static_assert(std::is_trivially_copyable_v<T>);

  size_t size() const {
    return end_ - data_;
  }

private:
  T* data_;
  T* end_;
  size_t capacity_;

  void reallocate(size_t new_cap){
    auto ndata = realloc(data_, new_cap * sizeof(T));
    if (!ndata) {
      throw std::bad_alloc();
    }
    capacity_ = new_cap;
    if (ndata != data_) {
      const auto old_size = size(); // Доступ к инвалидированному data_!
      data_ = ndata;
      end_ = data_ + old_size;
    } // Иначе OK, не операция
  }
}
```

Это код с неопределенным поведением. Скорее всего, оно никак не проявится сейчас, но это не значит, что так будет и впредь. Возможно, вызов `reallocate` заинлайнится в неподходящем месте, и всё пойдет кувырком.

Однако, приступая к реализации собственного вектора[2], надо учитывать следующий печальный факт: это невозможно сделать без неопределенного поведения (формального или реального). Основная причина — арифметика указателей внутри сырой памяти. В сырой памяти формально нет C++-массивов, только внутри которых арифметика и определена.

Размещающий оператор new и массивы

Вам посчастливилось добыть новую суперэффективную библиотеку для управления памятью? Вы хотите пользоваться ею в C++ и не сталкиваться с надуманным UB из-за проблем с лайфтаймами?

Вам повезло! Просто выделяйте память своей библиотекой, создавайте в выделенном буфере объекты с помощью оператора `new` (placement new) и забот не знайте!

```
void* buffer = my_external_malloc(sizeof(T), alignof(T));
auto pobj = new (buffer) T();
```

Красиво, просто, здорово!

А что, если мы захотим выделить память и разместить в ней массив?

Нет ничего проще!

```
void* buffer = my_external_malloc(n * sizeof(T), alignof(T));
auto pobjarr = new (buffer) T[n];
```

Всё, можно идти пить чай. Задача решена. Мы молодцы. Как похорошел C++ с 11-го стандарта!

Но не может же быть всё так просто?

Конечно же, нет! До C++20 вариант placement new для массивов имеет[3] полное право испоганить вашу память.

Конструкция `new (buffer) T[n];`, согласно примерам из стандарта C++17[4], переводится в:

```
operator new[](sizeof(T) * n + x, buffer);
// или operator new[](sizeof(T) * n + x,
//                    std::align_val_t(alignof(T)), buffer);
```

где *x* — никак не специфицируемое неотрицательное число, предназначенное, например, чтобы застолбить место под какую-либо метаинформацию о выделенном

[2] Стандартный вектор вполне может кого-то не устраивать тем, что он по умолчанию инициализирует память или использует исключения. Например, разработчики игр очень любят писать свой вектор.

[3] Если у вас старые версии компиляторов. Исправляющий defect report CWG 2382 (**https://cplusplus.github.io/CWG/issues/2382.html**) применили ретроспективно, ко всем версиям с C++98.

[4] См. стандарт C: § 8.5.2.4 на с. 110 (**https://clck.ru/3JLvFH**).

массиве: засунуть число элементов в начало области памяти или расставить маркеры начала/конца, или еще что-нибудь, что обычно делают аллокаторы.

То есть placement new для массива вполне может полезть за пределы предоставленного вами буфера. Очень удобно!

В C++20 восхитительную формулировку изменили.

Теперь же, если конструкция `new (arg1, arg2...) T[n];` соответствует вызову стандартного `void* operator new[](std::size_t count, void* ptr);`, то всё будет хорошо. Никаких магических сдвигов на *+x* не возникнет.

Но если же какой-то доброжелатель определил собственный operator placement new... Впрочем, это уже совсем другая история.

Я не встречал ни одного компилятора и ни одной поставки стандартной библиотеки, в которых бы стандартный placement new как-либо двигал указатель на пользовательский буфер. Реальную угрозу трудноотлавливаемого UB в большей степени представляют user-defined-версии placement new.

Чтобы обезопасить себя и вызвать настоящий стандартный placement new, нужно использовать `::new` и кастить указатель на буфер к `void*` либо положиться на алгоритмы `std::uninitialized_default_construct_n` и подобные ему.

Также нужно отметить, что в C++ нет синтаксиса placement delete. Мы можем только явно вызвать `operator delete[](void* ptr, void* place)`, стандартная версия которого ничего не делает.

Тут, конечно, нужно понимать разницу между самим `operator delete` и синтаксическими конструкциями `delete p` и `delete [] p`. Первый занимается только управлением памятью, последние же еще и вызывают деструкторы.

В C++ нет отдельной синтаксической конструкции, чтобы махом вызывать деструкторы элементов массива, созданного с помощью placement new. Это нужно делать вручную или использовать алгоритм `std::destroy`.

Ни в коем случае не стоит использовать `delete []` против указателя, полученного с помощью `placement new []`. Будет плохо.

Невыровненные ссылки

Программист форматировал байтики. Ведь это же самое любимое развлечение C++-программистов: писать снова и снова код для форматного вывода пользовательских структур.

Байтики у программиста были упакованными, чтоб никакого лишнего выравнивания! И поля у него были упорядочены также, чтоб никакого лишнего выравнивания.

```
#pragma pack(1)
struct Record {
  long value;
```

```
  int data;
  char status;
};

int main() {
  Record r { 42, 42, 42};
  static_assert(sizeof(r) == sizeof(int) + sizeof(char) + sizeof(long));
  std::cout <<
    std::format("{} {} {}", r.data, r.status, r.value); // 42 - '*'
}
```

Он проверял этот код с санитайзером, и санитайзер говорил ему, что всё в порядке.

```
Program returned: 0
42 * 42
```

Ну раз всё в порядке, то можно больше байтиков отформатировать!

```
int main() {
  Record records[] = { { 42, 42, 42}, { 42, 42, 42}  };
  static_assert(sizeof(records) ==
                2 * ( sizeof(int) + sizeof(char) + sizeof(long) ));
  for (const auto& r: records) {
    std::cout << std::format("{} {} {}", r.data, r.status, r.value); // 42 - '*'
  }
}
```

И что-то взорвалось (под ARM бы уж точно):

```
Program returned: 0
/app/example.cpp:16:48: runtime error: reference binding to
misaligned address 0x7ffd1eda9f85 for type 'const int',
which requires 4 byte alignment
0x7ffd1eda9f85: note: pointer points here
 00 00 00 00 2a 00 00
 00 2a 00 00 00 00 00
 00 00 00 00 00 00 00
 03 00 00 00 00 00 00  b0
```

Да, нельзя читать невыровненную память. Это влечет неопределенное поведение. Мы это уже знаем[5]. Нельзя разыменовывать невыровненный указатель. Но вот беда. В C++ же есть ссылки. И они тоже обязаны быть правильно выровненными.

Мы точно видим одну ссылку:

```
for (const auto& r: records);
```

[5] Если забыли — можно вернуться к разделу про `std::aligned_storage` в *главе 4*.

Но там же не тип const int! Ну да. Это Record, и с ней всё в порядке. #pragma pack(1) задает требование к выравниванию 1, так что тут никакой проблемы.

Откуда же взялась ссылка на const int?

А она у нас неявно взялась. Ведь неявное создание ссылок — это ключевая особенность C++!

```
template< class... Args >

// Вот они - эти два коварных &&!
std::string format( std::format_string<Args...> fmt, Args&&... args );

// Все три поля будут переданы по ссылке!
std::cout << std::format("{} {} {}", r.data, r.status, r.value);
```

Да, "универсальная ссылка" — это всё еще ссылка.

В упакованной структуре поля не выровнены. Ссылки на них брать нельзя.

Но ведь в первоначальном варианте с одной структурой работало же без предупреждений...

Ха! Нам просто повезло, что:

- поля в структуре упорядочены так, что и без *pragma pack* нет паддинга между ними;
- стек обычно выровнен на sizeof(void*), и этого достаточно для всех полей в структуре.

Мы можем добавить один лишний char на стек, и всё изменится.

```
int main() {
  char data[1];
  Record r { 42, 42, 42};
  memset(data, 0, 1);
  std::cout <<
    std::format("{} {} {}", r.data, r.status, r.value); // 42 - '*'
}

Program returned: 0
/app/example.cpp:17:44: runtime error:
reference binding to misaligned address 0x7ffe3b4e1f36 for type 'int',
which requires 4 byte alignment
0x7ffe3b4e1f36: note: pointer points here
 00 00 00 00 2a 00  00 00 2a 00 00 00 00 00
 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  00 00
```

Как же исправить это досадное недоразумение?

Нужно сделать отдельно чтение из каждого поля во временную правильно выровненную переменную, т. е. сделать копию.

```
int main() {
  Record records[] = { { 42, 42, 42}, { 42, 42, 42}  };
  for (const auto& r: records) {

    // В C++23 для этого есть замечательный auto().
    std::cout << std::format("{} {} {}",
      auto(r.data), auto(r.status), auto(r.value));

    // В C++20:
    auto data = r.data; auto status = r.status; auto value = r.value;
    std::cout << std::format("{} {} {}", data, status, r.value);

    // Или совершенно уродливо и неустойчиво к изменениям в типах:
    std::cout << std::format("{} {} {}", static_cast<int>(r.data),
                                         static_cast<char>(r.status),
                                         static_cast<long>(r.value>));
  }
}
```

В чуть более безопасных языках взятие невыровненных ссылок на поля упакованных структур просто не компилируется.

В Rust:

```
#[repr(C, packed)]
struct Record {
  value: i64,
  data: i32,
  status: i8,
}

fn main() {
  let r = Record { value: 42, data: 42, status: 42 };
  // В Rust макросы - одно из немногих мест,
  // где ссылки могут появляться неявно для читающего код.
  println!("{} {} {}", r.data, r.status, r.value);

  /*
  error[E0793]: reference to packed field is unaligned
  --> <source>:10:26
      |
```

```
   10 |     println!("{} {} {}", r.data, r.status, r.value);
      = note: packed structs are only aligned by one byte,
              and many modern architectures penalize unaligned
              field accesses
      = note: creating a misaligned reference is undefined
              behavior (even if that reference is never dereferenced)
      = help: copy the field contents to a local variable,
              or replace the reference with a raw pointer and
              use `read_unaligned`/`write_unaligned`
              (loads and stores via `*p` must be properly aligned
              even when using raw pointers)
  */

  // Вот так правильно:
  println!("{} {} {}", {r.data}, {r.status}, {r.value});
}
```

Правило strict aliasing и практика type punning

Особенно ревностные фанаты C и C++ любят говорить, что эти языки позволяют им всё контролировать. Даже малейшие прикосновения к памяти. Нужно просто научиться правильно пользоваться указателями и знать, как устроена память и как работают компьютеры! Байтики они всегда байтики... Вот только злобные разработчики компиляторов понапридумывают своих странных оптимизаций, которые ломают наш прекрасный и 100% правильный код! А когда не ломают, то ничего не оптимизируют!

Обсуждая правила алиасинга и неопределенного поведения при их нарушении, обычно демонстрируют какую-то странную, не очень реалистичную функцию от двух указателей и показывают (какой ужас!), что ее результат не соответствует ожиданиям при включении оптимизаций. Я, пожалуй, отойду от этой традиции и начну с функции, у которой никаких проблем нет.

```
// Не владеющий view над непрерывным массивом,
// например, для итерации по столбцу матрицы.
template <class T>
struct stride_view {
    T* data;
    size_t step;
    size_t len;

T& operator[](int idx) {
    return this->data[idx * step];
```

```
}
};
template <class T>
struct Data {
    stride_view<T> data;
    int counter;
};

void process(Data<uint32_t>* data, int idx) {
    data->data[idx] -= 1;
    if (data->data[idx] == 0) {
        data->counter -= 1;
    }
}
void process(Data<uint64_t>* data, int idx) {
    data->data[idx] -= 1;
    if (data->data[idx] == 0) {
        data->counter -= 1;
    }
}
```

Совершенно нормальные функции. Мы даже исполнять их не будем. Давайте их просто скомпилируем (GCC 14):

```
g++ -std=c++26 -O3
```

и посмотрим на сгенерированный код:

```
process(Data<unsigned int>*, int):
        movsx   rsi, esi
        imul    rsi, QWORD PTR [rdi+8]
        mov     rax, QWORD PTR [rdi]
        lea     rdx, [rax+rsi*4]
        sub     DWORD PTR [rdx], 1
        jne     .L1
        sub     DWORD PTR [rdi+24], 1
.L1:
        ret
process(Data<unsigned long>*, int):
        mov     rdx, QWORD PTR [rdi+8]
        movsx   rsi, esi
        mov     rax, QWORD PTR [rdi]
        sal     rsi, 3
        imul    rdx, rsi
```

```
        sub     QWORD PTR [rax+rdx], 1
        imul    rsi, QWORD PTR [rdi+8]
        cmp     QWORD PTR [rax+rsi], 0
        jne     .L4
        sub     DWORD PTR [rdi+24], 1
.L4:
        ret
```

И вот же странная картина! Версия с `uint32` явно оптимизирована чуть получше версии с `uint64`!

```
// Можно заметить, что код operator[]  (конкретно умножение на шаг)
// присутствует дважды,
// индекс пересчитывается дважды, и обращений к памяти два!
        imul    rdx, rsi
        sub     QWORD PTR [rax+rdx], 1
        imul    rsi, QWORD PTR [rdi+8]
        cmp     QWORD PTR [rax+rsi], 0
```

Ну, мы так и написали же в коде...

Да. Вот только в версии с `uint32` обращение лишь одно. А в коде на C++ же два!

```
        imul    rsi, QWORD PTR [rdi+8]
        mov     rax, QWORD PTR [rdi]
        lea     rdx, [rax+rsi*4]
        sub     DWORD PTR [rdx], 1
```

Обращения к памяти в основном очень дороги. А значит, от их оптимизации все будут только в плюсе! Зачем перечитывать повторно значение из памяти в регистр, если нам заранее известно, что оно там не поменялось?!

Но на основе чего можно получить такую информацию, чтоб иметь право выполнить оптимизацию?! Языки C и C++ же низкоуровневые. Полный контроль! Можно делать что угодно с памятью и указателями...

НЕТ. НЕЛЬЗЯ. Поприветствуем правила строгого алиасинга!

Если у вас есть указатели двух разных типов: `A* a` и `B* b`, — то запись значений через указатель `a` не влияет на чтение значений через указатель `b` во всех случаях, кроме нескольких исключений (алиасинг):

- `A` и `B` — это совместимые, `signed`/`unsigned`-версии одного и того же типа;
- `A` — это тип, совместимый с каким-либо из подобъектов внутри `B` (элемент структуры или объединения, элемент массива);
- `A` или `B` — `char`, `unsigned char` или `std::byte`. Этот вариант существует как раз для того, чтобы можно было работать с сырыми байтиками. Заметьте, что `signed char` не считается.

В C[6] допустимо еще:

- `A` и `B` — это совместимые структуры/объединения: у них совпадают размер, порядок и имена полей, а типы полей совместимы. Например `struct Vector { int32_t x; int32_t y; }` и `struct Point { uint32_t x; uint32_t y; }`.

Ну, звучит не страшно. Даже логично и правильно. Если у меня есть массив целых чисел, а еще рядом массив чисел с плавающей точкой, то в корректной программе от изменения числа в первом массиве во втором ничего не должно поменяться.

Теперь можно вернуться к примеру и понять, что пошло не так.

```
struct stride_view<uint32_t> {
    uint32_t* data;
    size_t step;
    size_t len;

uint32_t& operator[](int idx) {
    return this->data[idx * step];
}
};
...
void process(Data<uint32_t>* data, int idx) {
    data->data[idx] -= 1;        // Ссылка uint32_t&. Запись через нее.
    // Внутри stride_view нет полей типов int32.
    if (data->data[idx] == 0) {  // Запись выше не влияет на индекс,
                                 // и можно оптимизировать
        data->counter -= 1;
    }
}

struct stride_view<size_t> {
    size_t* data; // !!!
    size_t step;
    size_t len;

T& operator[](int idx) {
    return this->data[idx * step];
}
};
void process(Data<size_t>* data, int idx) {
    data->data[idx] -= 1;        // Ссылка size_t&. Запись через нее.
    // step имеет тип size_t. Ссылка теоретически может алиасить это поле!
```

[6] Не в C++, но компиляторы поддерживают!

```
    if (data->data[idx] == 0) {  // Запись выше влияет на индекс.
                                 // Нельзя оптимизировать!
        data->counter -= 1;
    }
}
```

Пытливый читатель, наверное, уже догадался, что случай с указателем `char*` совершенно восхитительно запрещает оптимизацию почти всегда. Ведь он может алиасить что угодно!

Строгий алиасинг — это особенность не только C и C++. Rust, например, тоже им следует, но еще более строго, включающая информацию об уникальных и разделяемых ссылках.

```
pub struct StrideView<'a, T> {
    pub data: &'a mut [T],
    pub step: usize,
}
impl <'a, T> StrideView<'a, T> {
    unsafe fn get<'b>(&'b mut self, idx: usize) -> &'b mut T {
            // unsafe, нас не интересует код обработки паник
        self.data.get_unchecked_mut(self.step * idx)
    }
}
pub struct Data<'a, T> {
    pub data: StrideView<'a, T>,
    pub counter: i32,
}
#[no_mangle]
pub unsafe fn process(data: &mut Data<'_, usize>, idx: usize) {
    *data.data.get(idx) -= 1; // Ссылка &mut usize
    // Она не может алиасить поле step,
    // так как иначе это означало бы, что data содержит ссылку на саму себя,
    // что невозможно для обычных ссылок в safe Rust.
    if (*data.data.get(idx) == 0) { // Можно оптимизировать!
        data.counter -= 1;
    }
}
process:
        imul    rsi, qword ptr [rdi + 16]
        mov     rax, qword ptr [rdi]
        dec     qword ptr [rax + 8*rsi]
        je      .LBB0_1
        ret
```

```
.LBB0_1:
        dec     dword ptr [rdi + 24]
        ret
```

Ну хорошо, существует такое правило. Оптимизировать позволяет. Отлично! А где неопределенное поведение?

А оно начинается тогда, когда разработчик творит безобразие с преобразованием указателей!

Вот теперь можно и классическую странную функцию показать!

```
#include <stdio.h>
int foo(int* x, short* y) {
    *x = 5;
    *y = 10;
    return *x;
}

int main() {
    int f = 0;
    printf("%d\n", foo(&f, (short*)&f));
}
```

В зависимости от уровня оптимизаций, на платформах little endian, результат будет `5` или `10`.

Функция, конечно, совершенно надуманная, но фокус в том, что она иллюстрирует простейший пример так называемого type punning — обращения с объектом одного типа, как с совершенно другим.

А вот он уже в реальных программах распространен! Быстрый обратный квадратный корень из Quake III — классический пример:

```
float Q_rsqrt( float number )
{
  int32_t i;
  float x2, y;
  const float threehalfs = 1.5F;

  x2 = number * 0.5F;
  y  = number;
  i  = * (int32_t * ) &y;     // Злобный хакинг на уровне чисел
                              // с плавающей точкой.
  i  = 0x5f3759df - ( i >> 1 );               // что за черт?
  y  = * ( float * ) &i;
  y  = y * ( threehalfs - ( x2 * y * y ) );   // первая итерация
//y  = y * ( threehalfs - ( x2 * y * y ) );   // вторая итерация, это можно удалить
  return y;
}
```

Такой лобовой каламбур типизации нарушает предположения строгого алиасинга. Нарушение объявлено неопределенным поведением. И результат можно получить самый странный.

Type punning очень распространен в библиотеках сериализации и десериализации для эффективного разбора сырых байтов. В реализации сетевых протоколов, например, всякие структуры `in_addr` используют `union` для type punning. В ядре Linux также можно найти примеры[7].

Что же делать?

Вся эта красота в любой момент может сломаться, если она сделана неправильно. Поэтому у GCC и Clang есть флаг `-fno-strict-aliasing`, чтоб вы могли отключить оптимизации на основе строгого алиасинга и позволить себе стрелять по ногам чуть более предсказуемо.

Type punning с помощью `reinterpret_cast` почти всегда выводит на поле неопределенное поведение. Язык C (не C++!) позволяет выполнять type punning с помощью `union`. В C++ до C++20 нужно было использовать `memcpy` и... копировать. Компиляторы при этом часто способны такие копии ради type punning оптимизировать и убирать. В C++20 добавили `std::bit_cast` для той же цели. В C++23 еще появится `start_lifetime_as`, который поможет для наиболее частого случая type punning: интерпретировать массив принятых байтов (например, по сети) как осмысленную структуру/массив структур.

Последнее, что нужно отметить: преобразование указателей само по себе не является неопределенным поведением. Так что, к примеру, стандартная цепочка `T* -> void* -> T*` при передаче объектов и функций между границами библиотек не нарушает правила алиасинга. А вот использование указателя `A*`, чтобы через него прочитать/изменить объект другого типа `B`... Проблемы начинаются здесь.

[7] Линус Торвальдс: *"This is why we use -fwrapv, -fno-strict-aliasing etc. The standard simply is not *important*, when it is in direct conflict with reality and reliable code generation"* ("Вот почему мы используем -fwrapv, -fno-strict-aliasing и т. д. Стандарт просто не имеет значения, когда он находится в прямом противоречии с реальностью и надежной генерацией кода").

ГЛАВА 7

Асинхронность и параллелизм

Гонка данных

Разработка многопоточных приложений — это всегда сложно. Проблема синхронизации доступа к разделяемым данным — вечная головная боль. Хорошо, если у вас уже есть оттестированная и проверенная временем библиотека контейнеров, высокоуровневых примитивов, параллельных алгоритмов, берущих на себя контроль за всеми инвариантами. Очень хорошо, если статические проверки компилятора не позволят вам использовать всё это добро неправильно. Ах, как было бы хорошо...

До C++11 и стандартизации модели памяти пользоваться потоками в принципе можно было лишь на свой страх и риск. Начиная с C++11, в стандартной библиотеке есть довольно низкоуровневые примитивы. С C++17 еще появились разные параллельные вариации алгоритмов, но о тонкой настройке количества потоков и приоритетов в них можете даже не думать.

Так почему бы не взять какую-нибудь готовую серьезную библиотеку (`boost`, `abseil`) — там наверняка умные люди уже пострадали многие часы, чтобы предоставить удобные и безопасные инструменты — и забот не знать?

Увы, так не работает. Правильность использования этих инструментов в C++ нужно контролировать самостоятельно, пристально изучая каждую строчку кода. Мы всё равно попадаем в проблемы синхронизации доступа с аккуратным развешиванием мьютексов и атомарных переменных.

Ситуация гонки данных (data race), в которой один поток программы модифицирует объект, а другой в то же самое время читает значения из этого объекта (или просто два потока одновременно пытаются модифицировать один объект), совершенно ясно является ошибочной. Результат чтения может выдать какой-то странный промежуточный объект, совместная запись — породить какое-то мутировавшее премешанное значение. И всё это независимо от языка программирования.

Но в C++ это не просто ошибка. Это неопределенное поведение. И "возможности" для оптимизации.

```
int func(const std::vector<int>& v) {
  int sum = 0;
  for (size_t i = 0; i < v.size(); ++i) {
    sum += v[i];
  }
```

```
  // Гонка данных запрещена, от модификации v в
  // параллельном потоке нас "защищает" неопределенное поведение.

  // А значит, можно соптимизировать вычисление size.
  // const size_t v_size = v.size();
  // for (size_t i = 0; i < v_size; ++i) { ... }
  return sum;
}
```

А теперь почти что многопоточный hello world:

```
int main() {
  bool terminated = false;
  using namespace std::literals::chrono_literals;

  int add_ms = 0;
  std::cin >> add_ms;

  std::jthread t1 { [&] {
    std::size_t cnt = 0;
    while (!terminated) {
      ++cnt;
    }
    std::cout << "count: " << cnt << "\n";
  } };

  std::jthread t2 { [&] {
    std::this_thread::sleep_for(500ms +
                                std::chrono::milliseconds(add_ms));
    terminated = true;
  } };
}
```

Мы не синхронизировали доступ всего лишь к какому-то bool. Ничего же страшного, ведь да? И в отладочной сборке всё работает.

Но если включить оптимизации, цикл в первом потоке либо не выполнит ни одной итерации (Clang), либо никогда не завершится (GCC)!

Оба компилятора видят, что доступ не синхронизирован. Data race запрещен. Значит, и синхронизировать не надо. Значит, при обращении к переменной terminate в заголовке цикла всегда должно быть одно и то же значение. GCC решает, что всегда будет false. Clang обнаруживает присваивание terminated = true в другом потоке и вытягивает его перед началом цикла.

Конечно же, тут ошибка намеренная и легко исправляется заменой bool на std::atomic<bool>. Но в реальной кодовой базе допустить гонку данных просто, а исправить сложнее.

Однажды я написал что-то подобное:

```
enum Task {
  done,
  hello
};
std::queue<Task> task_queue;
std::mutex mutex;

std::jthread t1 { [&] {
  std::size_t cnt_miss = 0;
  while (true) {
    if (!task_queue.empty()) {
      auto task = [&] {
        std::scoped_lock lock{mutex};
        auto t = task_queue.front();
        task_queue.pop();
        return t;
      }();
      if (task == done) {
        break;
      } else {
        std::cout << "hello\n";
      }
    } else {
      ++cnt_miss;
    }
  }
  std::cout << "count miss: " << cnt_miss << "\n";
} };

std::jthread t2 { [&] {
  std::this_thread::sleep_for(500ms);
  {
    std::scoped_lock lock{mutex};
    task_queue.push(done);
  }
} };
```

И оно прекрасно работало, пока код тестировался, будучи собранным одним компилятором. Но при переносе на другую платформу с другим компилятором всё сломалось — цикл обработки очереди перестал завершаться.

Если вы сразу поняли причину, то поздравляю. Иначе обратите внимание на безобидный метод `empty`, который "совершенно точно ничего не меняет" и порождает законное возмущение: "Да ладно, как там вообще может нарушиться консистентность данных?"

В поиске проблем с доступом к объектам из разных потоков вам помогут статические анализаторы и санитайзеры, например TSan для GCC/Clang (`-fsanitize=thread`). Но имейте в виду, что из-за особенности реализации санитайзеров ASan и TSan не могут работать одновременно. Так что не выйдет махом искать ими и условие гонки (race condition), и обычные ошибки доступа к памяти с нарушением времени жизни (lifetime).

В Rust нельзя создать data race и вызвать неопределенное поведение в безопасном подмножестве языка. Однако, неаккуратно используя `unsafe`, и в нем можно устроить себе проблемы. И будет неопределенное поведение. На то оно и `unsafe`.

Потокобезопасен ли указатель std::shared_ptr?

Пожалуй, это самый популярный вопрос при собеседовании на вакансию C++-разработчика. И не без причины: этим прекрасным умным указателем так просто пользоваться (в сравнении с его собратом `std::unique_ptr`), что легко не заметить подвох. В его названии есть `shared`. Да он и спроектирован так, чтобы его можно было разделять между потоками. Что может пойти не так?

Всё!

Новички довольно быстро обнаруживают первую линию костыльно-грабельной обороны бастиона сложности `shared_ptr`: если доступ к самому указателю `shared_ptr<T>` "безопасен", то к объекту `T` его все равно надо синхронизировать. Это очевидно, это заметно, это понятно. Но дальше ведь всё просто?

Нет.

Дальше притаились волчьи ямы с отравленными кольями. Сам объект-указатель `shared_ptr` не является потокобезопасным. И доступ к самому указателю тоже надо синхронизировать!

Как же так? Мы никогда не синхронизировали, и у нас всё работало.

Поздравляю, у вас одно из двух:

1. Либо все доступы к указателю из разных потоков осуществляют только чтение. И тогда проблем действительно нет.
2. Либо программа работает по воле случая.

```
using namespace std::literals::chrono_literals;
std::shared_ptr<std::string> str = nullptr;
std::jthread t1 { [&]{
  std::size_t cnt_miss = 0;
  while (!str) {
```

```
    ++cnt_miss;
  }
  std::cout << "count miss: " << cnt_miss << "\n";
  std::cout << *str << "\n";
} };

std::jthread t2 { [&] {
      std::this_thread::sleep_for(500ms);
      str = std::make_shared<std::string>("Hello World");
  }
};
```

Аналогично другим примерам с условием гонки код выше перестает работать при изменении уровня оптимизации.

Clang 19.1: `-std=c++20 -O3 -pthread`

```
Program terminated with signal: SIGSEGV
```

Но ведь вы наверняка что-то слышали: все-таки есть в `shared_ptr` кое-что потокобезопасное...

Да. Есть. Счетчик ссылок. Больше ничего потокобезопасного в `std::shared_ptr` нет. Атомарный счетчик ссылок как раз и позволяет без проблем копировать один и тот же указатель (увеличивая счетчики) в разные потоки и не синхронизировать вручную вызовы деструкторов (уменьшающих счетчики) в разных потоках.

Если вам надо менять указатель из разных потоков, вам нужен `std::atomic<std::shared_ptr<T>>` (C++20). Либо использовать функции `std::atomic_load/std::atomic_store` и прочие — у них есть специальные перегрузки для `shared_ptr`.

С `std::weak_ptr` все то же самое.

Ожидание потоков

А вы уже заметили, что в предыдущих главах я использую `std::jthread` из C++20 вместо `std::thread`? И зачем?

А всё очень просто: деструктор `std::thread` дурной.

Везде, где может начать вызываться деструктор `std::thread`, нужно втыкать:

```
// std::thread t1;
if (t1.joinable()) { // Если вы не уверены в богатой жизненной
                     // истории объекта t1,
                     // обязательно выполняйте эту проверку.
  t1.join(); // или t1.detach()
}
```

чтобы ознаменовать свое желание (или нежелание) дожидаться окончания выполнения потока. Иначе деструктор потока повалит вашу программу, вызвав `std::terminate`. Очень удобно и очень по RAII-шному, неправда ли?

Конечно, совсем везде втыкать приведенный фрагмент не надо. Он лишний, если:

- вы точно знаете, что кто-то другой уже выполнил это заклинание;
- или содержимое объекта `std::thread` переместили в другой объект (`t2 = std::move(t1)`).

И тем более не надо просто так втыкать этот код, обращающийся к одному и тому же объекту `std::thread` из разных потоков. Иначе — race condition. Надо синхронизировать.

И, конечно же, убедитесь, что этот код ни в коем случае не будет выполняться параллельно с вызовом деструктора `t1`. Деструктор тоже вызывает `joinable`, а это опять race condition.

Собираетесь сделать обертку над `std::thread`, чтобы вызывать `join` в ее деструкторе? Спешу порадовать: `join`/`detach` кидают исключения. Со всеми вытекающими проблемами[1].

Здорово, да? Поэтому в примерах был и будет `std::jthread`. Его деструктор сам выполняет `join` и снимает хотя бы часть головной боли.

А если вас `join` не устраивает, вы не хотите ждать и пользуетесь `detach`... Ну что ж. Ваше право. Только помните, что все потоки резко и внезапно помрут, когда закончится `main`.

Повторный захват мьютекса

Deadlock — это, конечно, печально. Система завязалась в узел и никогда не развяжется. А сколько мьютексов нужно, чтобы уйти в deadlock?

Немного подумав, можно решить, что одного достаточно — просто захвати его два раза подряд, не отпуская, в одном и том же потоке.

Возможно, под какой-то платформой это и так. Но в C++ это неопределенное поведение, и для красивого показательного дедлока нужны два мьютекса. А с одним ваш фокус не удастся и превратится в фокус от мира UB.

```
struct Test{
  std::mutex mutex;
  std::vector<int> v = { 1,2,3,4,5};

  auto fun(int n){
    mutex.lock();  // Захватываем.
    return
```

[1] Мы их обсуждали в *разд. "Ложный поехсерt" главы 5*.

```
      std::shared_ptr<int>(v.data() + n,
                           [this](auto...){mutex.unlock();});
                           // Освободим при смерти указателя.
  }
};

int main(){
  Test tt;
  auto a = tt.fun(1); // Захватили первый раз.
  std::cout << *a << std::endl;
  // Указатель жив.
  auto b = tt.fun(2); // Захватили второй раз. Неопределенное поведение.
  std::cout << *b << std::endl;

  return 0;
}
```

Этот пример дает разные результаты на одном и том же компиляторе, на одной и той же платформе, на одном и том же уровне оптимизаций. Все зависит от того, подключили pthread или нет.

Кто в здравом уме будет такое делать-то? Никто же никогда не захватывает один и тот же мьютекс два раза подряд.

Даже не знаю... Зачем-то же существуют рекурсивные мьютексы, которые можно захватывать по нескольку раз.

Да и сводить задачу к уже решенной и повторно использовать написанный код любят:

```
template <class T>
struct ThreadSafeQueue<T> {

bool empty() const {
  std::scoped_lock lock { mutex_ };
  ...
}

void push(T x) {
  std::scoped_lock lock { mutex_ };
  ...
}

std::optional<T> pop() {
  std::scoped_lock lock { mutex_ };
```

```
  if (empty()) { // ! ПОВТОРНЫЙ ЗАХВАТ !
    return std::nullopt;
  }
  ...
}

...
std::mutex mutex_;
};
```

Чтобы исправить, нужно либо подумать, либо использовать рекурсивный мьютекс. Но лучше подумать.

Методов у объекта может быть много. Разработчиков тоже. Они могут не помнить, где есть блокировка, а где — нет. Могут засунуть блокировку в один метод, забыв про другие. Так что от повторного захвата мьютекса в одном и том же потоке никто не застрахован.

Сигнало(не)безопасность

Разработчик любого сколько-нибудь серьезного приложения рано или поздно вынужден озаботиться вопросами поведения программы в различных краевых и внештатных ситуациях: запрос досрочного завершения, внезапное закрытие терминала, обработка маловероятных ошибочных состояний. Во многих этих случаях приходится иметь дело с довольно примитивным механизмом межпроцессного взаимодействия — с обработкой сигналов.

Программист регистрирует обработчики нужных ему сигналов и забот не знает, очень часто допуская серьезную ошибку, — выполняет в обработчике сигналов код, который там выполнять небезопасно: выделяет память, делает I/O, захватывает блокировки...

Сигналы прерывают нормальный ход исполнения программы и могут быть обработаны в произвольном потоке. Поток мог начать выделять память, захватить блокировку в аллокаторе и в этот момент быть прерванным сигналом. Если обработчик сигнала в свою очередь запросит выделение памяти... будет повторный захват блокировки в одном и том же потоке. Неопределенное поведение.

И результат может быть самым неожиданным. Например, в OpenSSH в 2006 году была обнаружена критическая уязвимость с возможностью удаленно получить root-доступ к системам с запущенным SSHD-сервером. Баг непосредственно связан с кодом, вызывавшим `malloc` и `free` при обработке сигналов. Уязвимость исправили, но в 2020 году, спустя 14 лет, ее случайно занесли обратно. Ошибку снова обнаружили и исправили лишь в 2024 году, и кто знает, сколько раз и кто воспользовался этой RegreSSHion за четыре года!

Упрощенно можно продемонстрировать проблему с сигналонебезопасным кодом на следующем примере:

```
std::mutex global_lock;

int main() {
  std::signal(SIGINT, [](int){
    std::scoped_lock lock {global_lock};
    printf("SIGINT!\n");
  });

  {
    std::scoped_lock lock {global_lock};
    printf("start long job\n");
    sleep(10);
    printf("end long job\n");
  }
  sleep(10);
}
```

Если мы скомпилируем эту программу под Linux (не забыв указать `-pthread`), запустим и нажмем <Ctrl>+<C>, то она навсегда зависнет из-за повторного захвата мьютекса одним и тем же потоком. Если же забудем `-pthread`, программа не зависнет и отработает "ожидаемым" образом.

Под Windows эта программа также работает "ожидаемо" из-за специфики обработки сигналов — там для обработки SIGINT/SIGTERM всегда неявно порождается новый поток.

В любом случае этот код некорректен из-за использования сигналонебезопасной функции внутри обработчика сигналов.

Обработка сигналов — вопрос крайне платформоспецифичный и зависит от конкретной прикладной задачи и архитектуры вашего приложения. Также это довольно сложный вопрос, если учитывать, что во время обработки одного сигнала нас могут прервать для обработки другого.

Наиболее часто встречаемое использование обработки сигналов — корректное завершение приложения с очисткой ресурсов и закрытием соединений. В общем, Graceful Shutdown (изящное завершение работы). В таком случае обычно обработка сигналов сводится к выставлению и проверке некоторого глобального флага.

Стандарты C и C++ описывают специальный целочисленный тип — `sig_atomic_t`. При доступе к переменным этого типа гарантируется сигналобезопасность. На практике он может оказаться просто алиасом для `int` или `long`. `volatile sig_atomic_t` можно использовать в качестве глобального флага, выставляемого в обработчике сигналов. Но только в однопоточной среде. Тут `volatile` необходим лишь для предотвращения нежелательной оптимизации — компилятор не делает предположений

о возможной обработке сигналов и прерывании нормального потока выполнения программы.

Нужно помнить, что `volatile` не дает гарантий потокобезопасности. И в многопоточной среде необходимо использовать настоящие атомарные типы, поддерживаемые на вашей платформе. Например, `std::atomic<int>`. Если, конечно, `std::atomic<T>::is_lock_free` истинно.

Как бороться?

- Делать обработчики сигналов как можно более простыми.
- Отключать автоматический прием сигналов и выполнять их обработку в рамках обычного исполнения программы (см., например, `sigprocmask` и `sigwait`).
- Сверяться с документацией, безопасно ли использование той или иной функции в контексте обработчика сигналов.
- Для флагов обработки сигналов использовать атомарные переменные, lock-free структуры или, если приложение однопоточное, `volatile sig_atomic_t`.

Примитив condition_variable, или Как сделать всё правильно и уйти в deadlock

Синхронизация потоков — это сложно, хотя у нас и есть примитивы. Такой себе каламбур. Хорошо, если есть готовые высокоуровневые абстракции в виде очередей или каналов. Но иногда приходится мастерить их самому с использованием более низкоуровневых вещей: мьютексов, атомарных переменных и обвязки вокруг них.

`condition_variable` — примитив синхронизации, позволяющий одному или нескольким потокам ожидать сообщений от других потоков. Ожидать пассивно, не тратя время CPU впустую на постоянные проверки чего-то в цикле. Поток просто снимается с исполнения, ставится в очередь операционной системой, а по наступлении определенного события (уведомления) от другого потока пробуждается. Всё замечательно и удобно.

Сам по себе примитив `condition_variable` не передает никакой информации, а только служит для пробуждения или усыпления потоков. Причем пробуждения из-за особенностей реализации блокировок могут случаться ложно, самопроизвольно (spurious), а не только по непосредственной команде через `condition_variable`.

Потому типичное использование требует дополнительной проверки условий и выглядит как-то так:

```
std::condition_variable cv;
std::mutex event_mutex;
bool event_happened = false;

// Исполняется в одном потоке.
void task1() {
```

```
  std::unique_lock lock { event_mutex };
  // Предикат гарантированно проверяется под
  // захваченной блокировкой
  cv.wait(lock, [&] { return event_happened; });
  // Беспредикатная версия wait ждет только уведомления,
  // но может произойти ложное пробуждение
  // (обычно если кто-то отпускает этот же мьютекс).
  ...
  // Дождались - событие наступило.
  // Выполняем, что нужно.
}

// Исполняется в другом потоке.
void task2() {
  ...
  {
    std::lock_guard lock {event_mutex};
    event_happened = true;
  }
  // Обратите внимание: вызов notify не обязан быть
  // под захваченной блокировкой. Однако в ранних
  // версиях MSVC, а также в очень старой версии из
  // boost были баги, требующие удерживать мьютекс
  // захваченным во время вызова notify().
  // Но есть случай, когда делать вызов notify
  // под блокировкой необходимо: если другой поток
  // может вызвать, например, завершаясь,
  // деструктор объекта cv.
  cv.notify_one(); // notify_all()
}
```

Хм, внимательный читатель может сообразить, что в функции task2 мьютекс используется только для защиты булевого флага. Невиданное расточительство! Целых два системных вызова в худшем случае. Давайте лучше флаг сделаем атомарным!

```
std::atomic_bool event_happened = false;
std::condition_variable cv;
std::mutex event_mutex;

void task1() {
  std::unique_lock lock { event_mutex };
```

```
  cv.wait(lock, [&] { return event_happened; });
  ...
}

void task2() {
  ...
  event_happened = true;
  cv.notify_one(); // notify_all()
  ...
}
```

Компилируем, запускаем, работает. Классно, срочно в релиз!

Но однажды приходит пользователь и говорит, что запустил `task1` и `task2` как обычно одновременно, но сегодня внезапно `task1` не завершается, хотя `task2` отработал! Вы идете к пользователю, смотрите — висит. Перезапускаете — не зависает. Еще перезапускаете — опять не зависает. Перезапускаете 50 раз — всё равно не зависает. "Сбой какой-то железный, разовый", — думаете вы.

Уходите. Через месяц пользователь опять приходит с той же проблемой. И опять не воспроизводится. Ну точно железный сбой, космическая радиация битик какой-то в локальном кеше треда выбивает. Ничего страшного...

На самом деле в программе ошибка, приводящая к блокировке при редком совпадении в порядке инструкций. Чтобы понять ее, нужно посмотреть внимательнее на то, как устроен метод `wait` с предикатом.

```
// Поток a
std::unique_lock lock {event_mutex};           // a1
// cv.wait(lock, [&] { return event_happened; }) это
while (![&] { return event_happened; }()) {    // a2
  cv.wait(lock);                               // a3
}

// ------------------------------
// Поток b
event_happened = true;      // b1
cv.notify_one();            // b2
```

Рассмотрим вот такую последовательность исполнения строчек в двух потоках:

1. `a1`: поток 1 захватывает блокировку.
2. `a2`: поток 1 проверяет условие — истинно, событие не наступило.
3. `b1`: поток 2 выставляет событие.
4. `b2`: уведомляет о нем, но поток 1 еще не начал ждать! Уведомление потеряно!
5. `a3`: поток 1 начинает ждать и никогда не дождется!

Отоптимизировали? Возвращайте захват мьютекса обратно.

Захват мьютекса в уведомляющем потоке гарантирует, что ожидающий уведомления поток либо еще не начал ждать и проверять событие, либо уже ждет его. Если же мьютекс не захвачен, мы можем попасть в промежуточное состояние.

Осторожнее с примитивами!

Гонки за vptr

Одна команда инженеров очень любила модель акторов. А еще эти инженеры очень любили писать на C++ и реализовывать всё с нуля, чтобы ни в коем случае не брать внешние зависимости. Поэтому они решили сделать собственную реализацию акторной модели.

Так сначала у них появился базовый класс:

```
class Actor {
public:
    virtual ~Actor() = default;
    // Мы опустим детали и объекты
    // для передачи сообщений между акторами.
    // Важно лишь что был метод run.
    virtual void run() = 0;
};
```

И этого было им достаточно, и довольно надолго... Пока внезапно не обнаружилось, что надо бы уметь запускать несколько акторов конкурентно и параллельно.

Тогда появился класс-наследник:

```
class AsyncActor: public Actor {
protected:
    virtual void RunImpl() = 0;

    void run() final {
        actor_thread_ = std::make_unique<std::thread>([this]{
            LOG_DEBUG("Started Asynchronously");
            this->RunImpl();
        })
    }
private:
    std::unique_ptr<std::thread> actor_thread_;
}
```

И всё было так же здорово, как и раньше. И все наследники `AsyncActor` работали исправно, как и было задумано. Да вот только некрасиво как-то это всё было! Метод

RunImpl вместо run нужно переопределять. Да и вообще все использования асинхронных акторов следовали паттерну

```
SomeAsyncActor actor{...};
actor.run();
```

run всегда надо было вызывать явно. А если его забыть, то ведь ничего не запустится же. А если запустить дважды, то произойдет страшное! Неудобно. Да еще и unique_ptr глаза мозолит... А что, если сделать по-умному?!

И тогда инженеры переписали AsyncActor:

```
// protected, чтобы больше не вызвать run извне.
class AsyncActor: protected Actor {
private:
    // Вау! Можно написать так красиво, и будет компилироваться!
    // Используем jthread, чтобы не думать про join.
    std::jthread actor_thread_ {
        [actor=this]{
            LOG_DEBUG("Started Asynchronously");
            actor->run();
        }
    };
};
```

И теперь достаточно было просто сделать

```
    class SomeAsyncActor : public SomeAsyncActor {
        void run() override {...}
    };

    SomeAsyncActor actor{...};
```

чтобы актор был успешно создан и сразу же запущен, как и требовалось всегда.

Команда была несказа́нно довольна таким изящным решением. Инженеры тестировали его долго и упорно вручную. И всё было отлично. На радостях они решили, что можно отключить печать отладочных логов на этапе компиляции. Так что строка LOG_DEBUG(...) превратилась в ничто.

Они пересобрали программу. Протестировали несколько раз. Всё продолжало работать. Они задеплоили приложение... И оно упало с ошибкой сегментации при старте. Открыв core dump, разработчики увидели: Pure virtual function called. Но ведь всё же работало?! Они включили логи обратно... И программа снова стала работать корректно.

В нашей истории оказалось аж два условия гонки (race conditions), спрятанных в самом неожиданном месте: в неявном обращении к указателю на таблицу виртуальных методов (vptr)!

В разделе про невиртуальные виртуальные функции *главы 5* мы уже обсуждали, что при вызове виртуального метода из конструктора или деструктора вызывается метод текущего класса, а не переопределенный.

В основных реализациях (Clang и GCC) виртуальных методов такой эффект достигается за счет переписывания указателя на таблицу виртуальных функций при входе в конструктор/деструктор и выходе из них.

Таким образом, если `actor_thread` вызовет `run` раньше, чем исполнение дойдет до конструктора `SomeAsyncActor`, будет вызван метод не того класса, который ожидается быть вызванным. Аналогично с деструктором. Ну а чего вы хотели, случайно вызывая методы на частично уничтоженном или еще не сконструированном объекте?! Это вообще-то неопределенное поведение.

Мы можем легко продемонстрировать эффект.

- Падение на деструкторе.

```
class SomeAsyncActor : public AsyncActor {
public:
    SomeAsyncActor() = default;
private:
    void run() override {
        printf("DO DO DO\n");
    }
};

int main() {
    SomeAsyncActor s;
    // Раскомментируйте, чтоб перестало падать.
    // std::this_thread::sleep_for(std::chrono::milliseconds(5));
}
```

Результат с Clang 18.1 `-O3 -std=c++20 -pthread`:

```
pure virtual method called
terminate called without an active exception
Program terminated with signal: SIGSEGV
```

- Падение на конструкторе.

```
...
class AsyncActor: protected Actor {
private:
    std::jthread actor_thread_ {
        [actor = this]{
            actor->run();
        }
    };
```

```
    // Искусственная задержка начала конструирования наследника.
    std::string metadata {
        (std::this_thread::sleep_for(std::chrono::milliseconds(1)), "metadata")
    };
};

class SomeAsyncActor : public AsyncActor {
public:
    SomeAsyncActor() = default;
private:
    void run() override {
        printf("DO DO DO\n");
    }
};

int main() {
    SomeAsyncActor s;
    std::this_thread::sleep_for(std::chrono::milliseconds(1));
}
```

Результат с Clang 18.1 `-O3 -std=c++20 -pthread`:

```
pure virtual method called
terminate called without an active exception
Program terminated with signal: SIGSEGV
```

Что ж, а на вопрос "Причем же тут отладочные логи?" я предлагаю теперь читателю ответить самостоятельно.

Асинхронное выполнение кода с помощью std::async

В каком-то виде поддержка асинхронного программирования появилась в C++11 вместе с моделью памяти и описанием поведения программы в многопоточной среде. И если посмотреть на предложенные возможности глазами разработчиков из 2011 года, то они даже в каком-то смысле могут показаться удачными... Однако в 2025 году мы имеем примеры куда более успешного дизайна как в других языках, так и в самом C++. Да, при всей монструозности, C++26 execution выглядит перспективно и, может быть, им даже научатся пользоваться правильно... Но C++26 еще не скоро станет стандартом по умолчанию в коммерческой разработке, так что вернемся к тому, что уже есть.

C++11 дал нам тип `std::future<T>` — примитив для отложенного (in future) получения результата типа `T`. Результат может вычисляться асинхронно, возможно, в дру-

гом потоке, возможно, просто отложено. Кто его знает — зависит от конкретной реализации вычислений, которые вам этот `std::future` выдали в качестве обещания результата.

И вроде бы всё здорово. Но есть нюанс, который разочарует любого, кто поработал с похожими сущностями из других языков — с Promise из JavaScript или с Future из Rust, — эргономичность `std::future` совершенно ужасна. Если вы привыкли к монадическим цепочкам `map`, `and_then`, отвыкайте! `std::future` не поддерживает их. Вы можете только синхронно ждать результат. Вы хотите ждать результат сразу нескольких вычислений и отреагировать на любой из них? Такой роскоши тоже нет в стандартной библиотеке. Существует, конечно, `std::experimental::when_any`, но это, очевидно, экспериментальная функция.

Тем не менее для не очень серьезных приложений `std::future`, может, и сгодится даже сегодня. Давайте, например, напишем простенький сервер, который будет принимать запросы (мы не ожидаем большой нагрузки) и исполнять их асинхронно. Если вы подумали, что для этого нам сейчас понадобится сделать собственный пул потоков... то вы, конечно, правильно подумали, но C++11 предлагает нам готовое решение — `std::async`.

```
#include <future>
#include <iostream>
#include <thread>
#include <chrono>

struct Request { std::string data; };
struct Response { std::string data; };

Request accept_request(int i) {
    std::string data = "request " + std::to_string(i);
    return {data};
}

Response process_request(Request r) {
    // Имитируем ввод/вывод.
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    return { "processed: " + r.data + "\n" };
}

int main() {
    for (int i = 0; i < 5; ++i) {
        auto r = accept_request(i);
        std::async([r = std::move(r)]() mutable {
```

```
            auto response = process_request(std::move(r));
            std::cout << response.data;
        });
    }
}
```

Компилируем с ключом `-std=c++14`, запускаем:

```
processed: request 0
processed: request 1
processed: request 2
processed: request 3
processed: request 4
```

Вроде всё верно? А сколько времени должен исполняться этот цикл? Измерим:

```
int main() {
    auto start = std::chrono::steady_clock::now();
    for (int i = 0; i < 5; ++i) {
        auto r = accept_request(i);
        std::async([r = std::move(r)]() mutable {
            auto response = process_request(std::move(r));
            std::cout << response.data;
        });
    }
    auto end = std::chrono::steady_clock::now();
    std::cout << "elapsed: " <<
std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count() << "\n";
}
```

500 миллисекунд. Никакого совпадения. Пять запросов. На каждый по 100 миллисекунд. Как заказывали. Да, наш код оказался никаким не асинхронным. Обработка выполняется строго последовательно.

Неожиданно? Если вы никогда не видели C++, но при этом написали множество асинхронного кода на Rust или Go, то, думаю, для вас это станет неожиданным результатом. Ведь с `go routines` и `tokio::spawn` таких неприятностей не было.

Если мы соберем этот код с ключом `-std=c++17`, то обнаружим предупреждение компилятора!

```
<source>: In function 'int main()':
<source>:25:19: warning: ignoring return value of 'std::future<typename
std::__invoke_result<typename std::decay<_Tp>::type, typename std::decay<_Args>::type
...>::type> std::async(_Fn&&, _Args&& ...) [with _Fn = main()::<lambda()>; _Args = {};
typename __invoke_result<typename decay<_Tp>::type, typename decay<_Args>::type
...>::type = void; typename decay<_Tp>::type = main()::<lambda()>]', declared with
attribute 'nodiscard' [-Wunused-result]
   25 |         std::async([r = std::move(r)]() mutable {
      |         ~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
   26 |             auto response = process_request(std::move(r));
      |             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   27 |             std::cout << response.data;
      |             ~~~~~~~~~~~~~~~~~~~~~~~~~~
   28 |         });
```

Да-да. И оно непосредственно относится к нашей проблеме. Мы проигнорировали результат `std::async`. А результатом была `std::future<T>`. У проигнорированной `std::future` естественным образом вызывается деструктор. Ну-ка, что про него пишут?

Начиная с C++14 есть приписка:

`These actions will not block for the shared state to become ready, except that they may block if all following conditions are satisfied:`

`The shared state was created by a call to std::async.`

`The shared state is not yet ready.`

`The current object was the last reference to the shared state.`

В этом конкретном специальном случае, когда `std::future<T>` создана с помощью `std::async`, у нее блокирующий деструктор! И это даже в каком-то смысле безопасно: ваша "асинхронная" задача в фоновом потоке не будет прервана внезапно и безрезультатно, если основной поток программы достигнет конца `main`-функции.

Так что мы должны внести изменения и собрать полученные `std::future` в какой-нибудь контейнер:

```
int main() {
    std::list<std::future<void>> pending_tasks;
    auto start = std::chrono::steady_clock::now();
    for (int i = 0; i < 5; ++i) {
        auto r = accept_request(i);
        pending_tasks.push_back(std::async([r = std::move(r)]() mutable {
            auto response = process_request(std::move(r));
            std::cout << response.data;
        }));
    }
    // Сбрасываем все и ждем.
    { auto _ = std::move(pending_tasks); }
    auto end = std::chrono::steady_clock::now();
    std::cout << "elapsed: " <<
std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count() << "\n";
}
```

И вот теперь у нас получен ожидаемый результат:

```
processed: request 0
processed: request 3
processed: request 1
```

```
processed: request 4
processed: request 2
elapsed: 100
```

Победа? Не совсем. Мы всё еще продолжаем неправильно использовать `std::async`! При вызове его в такой форме поведение не специфицировано. Есть два варианта.

- Переданная функция каким-то образом начнет исполняться асинхронно (в фоновом потоке или в пуле потоком — это тоже не специфировано). И такое поведение по умолчанию во всех современных версиях GCC, Clang и MSVC.
- Переданная функция не будет выполняться до тех пор, пока вы не вызовете `wait` или `get` у возвращенной `std::future`. И такое поведение долгое время было со старыми версиями компиляторов, например GCC 5.4.

Какое именно поведение можно и нужно контролировать с помощью вызова перегрузки `std::async` с дополнительным первым параметром типа `std::launch`? А вот такое:

- `std::launch::async` — если вы действительно хотите асинхронное исполнение;
- `std::launch::deferred` — если нужно отложить до точки вызова `wait` или `get`.

Пожалуй, это одна из изысканных шуток стандартной библиотеки C++: `std::async(f)` — по умолчанию не `async`, а `std::async(std::launch::async | std::launch::deferred, f)`.

И правильный код будет выглядеть так:

```
int main() {
    std::list<std::future<void>> pending_tasks;
    ...
    for (int i = 0; i < 5; ++i) {
        auto r = accept_request(i);
        pending_tasks.push_back(std::async(std::launch::async,
            [r = std::move(r)]() mutable {
                auto response = process_request(std::move(r));
                std::cout << response.data;
            }));
    }
    // Сбрасываем все и ждем.
    { auto _ = std::move(pending_tasks); }
    ...
}
```

Теперь точно победа? Да, но только в таком совсем простом примере... Ведь в реальности же ваш сервер будет принимать нефиксированное количество запросов. А значит, список из `std::future` будет расти... и расти... и расти... И вам все-таки придется взять и самостоятельно запустить фоновый поток, который будет их потихоньку из списка выбрасывать. Считайте это домашним заданием. После его вы-

полнения можете ответить на вопрос: *"Стоит ли использовать std::async, чтобы не думать о собственном пуле потоков?"*

Конкурентный доступ к файловой системе

Условия гонки за ресурсы могут возникать на разных уровнях:

- внутри одной программы;
- между несколькими программами на одном компьютере;
- между разными программами на разных компьютерах.

И почти всегда это нежелательная, ошибочная ситуация, последствия которой варьируются от просто неправильного результата до чудовищных уязвимостей в безопасности.

Иногда, конечно, система может быть толерантной к таким ошибкам, и серьезных проблем не возникнет: например, если страничка магазина запрашивает список товаров на складе, а в этот момент база данных склада обновляется, то вы, как пользователь, возможно, увидите неполный или устаревший список товаров или смесь из старых и новых данных. Но критического ничего не произойдет, если разработчики предусмотрели дополнительные проверки с запросом к базе данных в момент взаимодействия с конкретным товаром из списка... Чтобы вы не купили то, чего больше не существует.

Файловая система — один из таких ресурсов, гонки за которым естественны и должны учитываться при разработке приложений. Да, самые низкоуровневые проблемы синхронизации чтения и записи в файловую систему берет на себя операционная система и (или) конкретный драйвер. Можно "спокойно" одновременно из разных процессов читать и писать в один и тот же файл и получать мусор или штатные ошибки: низкоуровневые операции `read` и `write` будут как-то упорядочены планировщиком ввода-вывода. С самой файловой системой при этом всё будет в порядке.

Но высокоуровневые операции над файловой системой в рамках вашей бизнес-логики требуют внимания и осторожности. Ведь самый простой способ получить условие гонки — добавить проверки при открытии файла!

```
#include <filesystem>
#include <string>
#include <fstream>

namespace fs = std::filesystem;

int main() {
    if (!fs::exists("/my/file")) {
        return EXIT_FAILURE;
    }
```

```
    std::ifstream input("/my/file");
    std::string line; input >> line;
    // Что-то делаем.
    return EXIT_SUCCESS;
}
```

Это классическая ошибка TOCTOU (Time-of-Check-Time-of-Use). Между проверкой и открытием файла собственно файл может быть удален.

Но причем тут C++, если такая проблема существует для любого языка программирования?

Действительно. Например, во всех версиях стандартной библиотеки Rust с 1.0 до 1.58.1 была похожая ошибка в реализации функции `remove_dir_all`.

```
// Это упрощенный код.
// directory: Path
if directory.is_symlink() {
    remove_link(directory)
} else {
    remove_recursive(directory)
}
```

Между проверкой и удалением злоумышленник мог подменить настоящий каталог на символьную ссылку и добиться удаления данных, к которым у него нет доступа.

Ошибку исправили: вместо работы с путями функция стала работать с элементами в каталоге через единожды открытываемый файловый дескриптор.

А теперь мы можем вернуться к C++.

В `std::filesystem` также есть функция `remove_all`, с тем же значением, что и версия из Rust. И в большинстве ее реализаций еще в 2022 году также нашли точно такую же ошибку! Обсуждение этой ошибки было довольно бурным, поскольку стандарт C++ объявляет неопределенным поведением любые вызовы функций `std::filesystem`, приводящие к гонке!

Но они все приводят к гонке! Их корректная работа зависит только от благонадежности других приложений, обращающихся к той же файловой системе. Можно ли в таком случае ничего не исправлять в функции `std::filesystem::remove_all`?

Конечно, нужно! Ошибка была исправлена в GCC версии 11.5 и в Clang 14.

Послесловие: статический анализ и неопределенное поведение

Неопределенное поведение стараются искать с помощью статического анализа кода, но очень сложно сформулировать, что именно должны делать анализаторы. Нет конкретного паттерна/описания, что, собственно, нужно найти. Неопределенное поведение — это набор разнообразнейших способов написать неправильный код. Более того, на планете UB дрейфуют целые континенты ошибок, которые обычно в статических анализаторах рассматриваются отдельно: разыменование нулевых указателей, переполнение знаковых чисел, выход за границы буферов и т. д.

Для разработчиков анализаторов поиск UB — это одновременно сложная и интересная задача. Здесь недостаточно использовать какую-то одну технологию. Анализ потока данных поможет выявить использование невалидного указателя, но перпендикулярен задаче выявления использования зарезервированных имен. Поиск UB — это выявление множества отдельных модельных вариантов ошибок.

Другими словами, поиск UB разветвляется на поиск разнородных паттернов, закономерностей и частных случаев. Именно поэтому Андрей Карпов и команда PVS-Studio с таким интересом поучаствовали в подготовке этой книги. В анализаторе уже много сделано для поиска UB, но и еще столько же предстоит сделать. Эта книга станет путеводной звездой и источником вдохновения.

Спасибо Дмитрию от команды PVS-Studio!

Интернет-источники и литература

(char)0 = 0; - What Does the C++ Programmer Intend With This Code? - JF Bastien - C++ on Sea 2023 // YouTube : site. — URL: https://www.youtube.com/watch?v=dFIqNZ8VbRY.

/permissive- (Standards conformance) // MSVC documentation. Microsoft : site. — URL: https://clck.ru/3JRESK.

A Plan for C++26 Ranges. — URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2760r0.html/.

AddressSanitizer // Clang 21.0.0git documentation : site. — URL: https://clang.llvm.org/docs/AddressSanitizer.html.

alignas specifier // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/language/alignas.

Almost Always Auto : blog. — URL: http://cginternals.github.io/guidelines/articles/almost-always-auto/.

Assencio D. Avoid using floating-point numbers as hash table keys : blog. — URL: https://dassencio.org/98.

Assigning to enable_shared_from_this::__weak_this twice. — URL: https://cplusplus.github.io/LWG/issue2529.

Bancila M. C++ code samples before and after Ranges // Marius Bancila's Blog : site. — URL: https://mariusbancila.ro/blog/2019/01/20/cpp-code-samples-before-and-after-ranges/.

Bancila M. Little-known C++: function-try-block // Marius Bancila's Blog : site. — URL: https://mariusbancila.ro/blog/2019/03/13/little-known-cpp-function-try-block/.

Baptiste Wicht. C++ benchmark - std::vector VS std::list : blog. — URL: https://baptiste-wicht.com/posts/2012/11/cpp-benchmark-vector-vs-list.html.

Belle Views on C++ Ranges, their Details and the Devil - Nico Josuttis - Keynote Meeting C++ 2022 // YouTube : site. — URL: https://www.youtube.com/watch?v=O8HndvYNvQ4.

Boccara J. The Most Vexing Parse: How to Spot It and Fix It Quickly // Fluent C++ : site. — URL: https://www.fluentcpp.com/2018/01/30/most-vexing-parse/.

Boehm garbage collector // Wikipedia : site. — URL: https://en.wikipedia.org/wiki/Boehm_garbage_collector.

bugprone-use-after-move // Extra Clang Tools 21.0.0git documentation : site. — URL: https://clang.llvm.org/extra/clang-tidy/checks/bugprone/use-after-move.html.

C numeric limits interface // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/types/climits.

C: unary minus operator behavior with unsigned operands // Stack Overflow : site. — URL: https://clck.ru/3JTBkw.

C++ attribute: no_unique_address // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/language/attributes/no_unique_address.

C++ attribute: noreturn // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/language/attributes/noreturn.

C++ FAQ. Garbage collection ABI. — URL: https://isocpp.org/wiki/faq/cpp11-library#gc-abi.

C++ Standard. Integer promotion. — URL: https://eel.is/c++draft/conv.prom.

C++ Standards Committee Papers. Minimal Support for Garbage Collection and Reachability-Based Leak Detection. — URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2670.htm.

C++ Standards Committee Papers. Removing Garbage Collection Support. — URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2186r0.html.

Carlini N., Barresi A., Payer M., David W., Gross T. R. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity / 24th USENIX Security Symposium ; August 12–14, 2015 ; Washington, D.C. — URL: https://goo.su/oK4HMNp.

Copy elision // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/language/copy_elision.

Corbet J. Variable-length arrays and the max() mess // LWN.net : site. — URL: https://lwn.net/Articles/749064/.

Coroutines // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/language/coroutines.

CP.51: Do not use capturing lambdas that are coroutines // C++ Core Guidelines : site. — URL: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rcoro-capture.

CppCon 2019: Chandler Carruth "There Are No Zero-cost Abstractions" // YouTube : site. — URL: https://www.youtube.com/watch?v=rHIkrotSwcc&t=1065s.

Cro L. Struct of Arrays (SoA) in Zig : blog. — URL: https://zig.news/kristoff/struct-of-arrays-soa-in-zig-easy-in-userland-40m0.

CWE-190: Integer Overflow or Wraparound // CWE : site. — URL: https://cwe.mitre.org/data/definitions/190.html.

Dargo S. C++23: auto(x) and decay copy // Sandor Dargo's Blog : site. — URL: https://www.sandordargo.com/blog/2022/11/30/cpp23-auto-and-decay-copy.

DCL51-CPP. Do not declare or define a reserved identifier // SEI CERT C++ Coding Standard : site. — URL: https://clck.ru/3JSv5q.

decltype (C++) // Microsoft. MSDN : site. — URL: https://learn.microsoft.com/ru-ru/cpp/cpp/decltype-cpp?view=msvc-170.

Definitions and ODR (One Definition Rule) // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/language/definition.

Deprecate std::aligned_storage and std::aligned_union. — URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p1413r3.pdf.

Do condition variables still need a mutex if you're changing the checked value atomically? // Stack Overflow : site. — URL: https://clck.ru/3JSzPK.

Empty base optimization // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/language/ebo.

enable_shared_from_this // Cppreference.com : site. — URL: https://clck.ru/3JSzzD.

EXP61-CPP. A lambda object must not outlive any of its reference captured objects // SEI CERT C++ Coding Standard : site. — URL: https://clck.ru/3JS5Bj.

EXP63-CPP. Do not rely on the value of a moved-from object // SEI CERT C++ Coding Standard : site. — URL: https://clck.ru/3JS8dn.

explicit specifier // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/language/explicit

Extending the namespace std. Designated addressable functions // Cppreference.com : site. — URL: https://clck.ru/3JSczc.

Fast inverse square root // Wikipedia : site. — URL: https://en.wikipedia.org/wiki/Fast_inverse_square_root.

Fasterthanlime. A Rust match made in hell : blog. — URL: https://fasterthanli.me/articles/a-rust-match-made-in-hell.

Filipek B. C++ Initialization story: a guide through all initialization options and related C++ areas (C++ stories). — 294 p.

Function template // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/language/function_template.

Function try block // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/language/try#Function_try_block.

GCC Easter Egg: C++ Undefined Defined Behavior : blog. — URL: https://feross.org/gcc-ownage/.

gMock for Dummies // GooglTest : site. — URL: https://google.github.io/googletest/gmock_for_dummies.html

Great C++ is_trivial - Jason Turner - CppCon 2019 // YouTube : site. — URL: https://www.youtube.com/watch?v=ZxWjii99yao.

Griffing D. MSVC C++20 and the /std:c++20 Switch // Microsoft. Dev Blogs : site. — https://devblogs.microsoft.com/cppblog/msvc-cpp20-and-the-std-cpp20-switch/.

Grimm R. C++ Core Guidelines: Be Aware of the Traps of Condition Variables // MC++ BLOG & NEWS : site. — URL: https://clck.ru/3JSjNu

Grimm R. C++ Core Guidelines: The noexcept Specifier and Operator // MC++ BLOG & NEWS : site. — URL: https://clck.ru/3JSj9W.

Grimm R. C++20: The Ranges Library // MC++ BLOG & NEWS : site. — URL: https://www.modernescpp.com/index.php/c-20-the-ranges-library/

Gruevski Predrag. Ложные представления программистов о неопределенном поведении // PV-Studio : сайт. — URL: https://pvs-studio.ru/ru/blog/posts/cpp/1024/.

Hooking on Linux with LD_PRELOAD. — URL: https://liveoverflow.com/hooking-on-linux-with-ld_preload-pwn-adventure-3/.

How do I remove code duplication between similar const and non-const member functions? // Stack Overflow : site. — URL: https://clck.ru/3JSK7y.

IEEE 754 // Wikipedia : site. — URL: https://en.wikipedia.org/wiki/IEEE_754.

Implicit type promotion rules // Stack Overflow : site. — URL: https://stackoverflow.com/questions/46073295/implicit-type-promotion-rules.

Implicit type promotion rules // Stack Overflow : site. — URL: https://clck.ru/3JT2nB.

INT32-C. Ensure that operations on signed integers do not result in overflow // SEI CERT C Coding Standard : site. — URL: https://clck.ru/3JT2C4.

Is a moved-from vector always empty? Answer // Stack Overflow : site. — URL: https://stackoverflow.com/questions/17730689/is-a-moved-from-vector-always-empty/17735913#17735913.

ISO C++ FAQ. What is "const correctness"? — URL: https://isocpp.org/wiki/faq/const-correctness.

Iterators and Ranges: Comparing C++ to D to Rust - Barry Revzin - [CppNow 2021] // YouTube : site. — URL: https://www.youtube.com/watch?v=d3qY4dZ2r4w.

Josuttis N. et. al. WG21 D2012R0. Heal C++: Fix the range-based for loop, Rev0. — URL: https://josuttis.com/download/std/D2012R0_fix_rangebasedfor_201029.pdf.

Kapil D. Buffer Overflow Exploit : blog. — URL: https://dhavalkapil.com/blogs/Buffer-Overflow-Exploit/.

Kline M. Comparing Floating-Point Numbers Is Tricky : blog. — URL: https://bitbashing.io/comparing-floats.html.

Lambda expressions // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/language/lambda.

Lazy Evaluation and Aliasing // Eigen : site. — URL: https://eigen.tuxfamily.org/dox/TopicLazyEvaluation.html.

Linux manual page. alloca // man7.org : site. — URL: https://man7.org/linux/man-pages/man3/alloca.3.html.

Linux manual page. signal-safety // man7.org : site. — URL: https://man7.org/linux/man-pages/man7/signal-safety.7.html.

List of tools for static code analysis // Wikipedia : site. — URL: https://clck.ru/3JTE7A.

LiveOverflow. A simple Format String exploit example - bin 0x11 // YouTube : site. — URL: https://www.youtube.com/watch?v=0WvrSfcdq1I.

Memcpy (and friends) with NULL pointers // Hacker News. — URL: https://news.ycombinator.com/item?id=12002746.

Modern (Effective) C++ - Avoid default capture modes // cppatomic : site. — URL: https://cppatomic.blogspot.com/2018/03/modern-effective-c-avoid-default.html.

Müller J. Move Safety — Know What Can Be Done in the Moved-From State : blog. — URL: https://www.foonathan.net/2016/07/move-safety/.

noexcept operator // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/language/noexcept.

noexcept specifier // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/language/noexcept_spec.

Nystrom B. What color is your function : blog. — URL: https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/.

O'Dwyer A. On lifetime extension and decltype(auto) // Stuff mostly about C++ : blog. — URL: https://quuxplusone.github.io/blog/2018/04/05/lifetime-extension-grudgingly-accepted/.

O'Neal B. Using C++17 Parallel Algorithms for Better Performance // Microsoft. Dev blogs. — URL: https://devblogs.microsoft.com/cppblog/using-c17-parallel-algorithms-for-better-performance/.

Options to Request or Suppress Warnings // GCC documentation : site. — URL: https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html.

P1957R2. Converting from T* to bool should be considered narrowing. — URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1957r2.html.

Pattis R.E. Self-Referential Classes/Linked Objects: advanced programming/practicum // 15-200 : site. — URL: https://www.cs.cmu.edu/~pattis/15-1XX/15-200/lectures/linkedlists/index.html.

Pieterse V., Kourie D.G., Cleophas L., Watson B.W. Performance of C++ bit-vector implementations // ResearchGate : site. — URL: https://www.researchgate.net/publication/220803585_Performance_of_C_bit-vector_implementations.

Range-based for loop // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/language/range-for.

Ranges library // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/ranges.

Reference initialization. Lifetime of a temporary // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/language/reference_initialization#Lifetime_of_a_temporary.

Regehr J. A Guide to Undefined Behavior in C and C++ : blog. — URL: https://blog.regehr.org/archives/213.

Region Based memory management // Wikipedia : site. — URL: https://en.wikipedia.org/wiki/Region-based_memory_management.

Reis G.D, Sutter H., Caves J. Refining Expression Evaluation Order for Idiomatic C++. — URL: https://open-std.org/Jtc1/sc22/wg21/docs/papers/2016/p0145r1.pdf.

Ropert M. Copy and Swap, 20 years later : blog. — URL: https://mropert.github.io/2019/01/07/copy_swap_20_years/.

Saini M. Shocking Examples of Undefined Behaviour : blog. — URL: https://clck.ru/3JTCpA.

Shafik Yaghmour. What is the Strict Aliasing Rule and Why do we care // GitHub : site. — URL: https://clck.ru/3JTBJ5.

Sign Extension // Sun Studio 12: C User's Guide : site. — URL: https://docs.oracle.com/cd/E19205-01/819-5265/bjamz/index.html.

signal // Microsoft C++, C, and Assembler documentation : site. — URL: https://clck.ru/3JSz8R.

Šimon Tóth. C++20 Ranges — Полное руководство // PV-Studio : сайт. — URL: https://pvs-studio.ru/ru/blog/posts/cpp/0895/.

Smith L. Rust Functions Are Weird (But Be Glad) // YouTube : site. — URL: https://www.youtube.com/watch?v=SqT5YglW3qU.

Stack Overflow. Iterator invalidation rules for C++ containers // Stack Overflow : site. — URL: https://stackoverflow.com/questions/6438086/iterator-invalidation-rules-for-c-containers.

std::allocator // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/memory/allocator.

std::array<T,N>::operator[] // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/container/array/operator_at.

std::atomic<std::shared_ptr> // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/memory/shared_ptr/atomic2.

std::isnan // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/numeric/math/isnan.

std::jthread // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/thread/jthread.

std::make_shared, std::make_shared_for_overwrite // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/memory/shared_ptr/make_shared.

std::ranges::for_each, std::ranges::for_each_result // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/algorithm/ranges/for_each.

std::same_as // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/concepts/same_as.

std::unreachable_sentinel_t, std::unreachable_sentinel // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/iterator/unreachable_sentinel_t.

std::void_t // Cppreference.com : site. — URL: https://en.cppreference.com/w/cpp/types/void_t.

STL Concepts and Ranges // YouTube : site. — URL: https://www.youtube.com/watch?v=8yV2ONeWXyI.

Streaming SIMD Extensions // Wikipedia : site. — URL: https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions.

Sutter H. Move, simply : blog. — URL: https://herbsutter.com/2020/02/17/move-simply/.

Tessil. Benchmark of major hash maps implementations. — URL: https://tessil.github.io/2016/08/29/benchmark-hopscotch-map.html.

The downsides of C++ Coroutines // Reductor's blog : site. — URL: https://reductor.dev/cpp/2023/08/10/the-downsides-of-coroutines.html.

The GNU C Library (glibc) manual. Reserved Names. — URL: https://www.gnu.org/software/libc/manual/html_node/Reserved-Names.html.

The GNU C Library. Blocking Signals. — URL: https://www.gnu.org/software/libc/manual/html_node/Blocking-Signals.html.

ThreadSanitizer // Clang 21.0.0git documentation : site. — URL: https://clang.llvm.org/docs/ThreadSanitizer.html.

To what degree does std::shared_ptr ensure thread-safety? // Stack Overflow : site. — URL: https://clck.ru/3JSyYt.

Torvalds L. Torvalds on aliasing. — URL: https://www.yodaiken.com/2018/06/07/torvalds-on-aliasing/.

Undefined, unspecified and implementation-defined behavior // Stack Overflow : site. — URL: https://stackoverflow.com/questions/2397984/undefined-unspecified-and-implementation-defined-behavior.

UndefinedBehaviorSanitizer // Clang 21.0.0git documentation : site. — URL: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

Unit in the last place // Wikipedia : site. — URL:https://en.wikipedia.org/wiki/Unit_in_the_last_place.

va_list // Cppreference.com : site. — URL: https://en.cppreference.com/w/c/variadic/va_list.

Valappil D. C++ Pitfalls — std::move is NOT moving anything! : blog. — URL: https://medium.com/@dhaneshvb/c-pitfalls-std-move-is-not-moving-anything-c9c073422b83.

Watters B. Stack-Based Buffer Overflow Attacks: Explained and Examples : blog. — URL: https://clck.ru/3JSjiS.

Weak symbol // Wikipedia : site. — URL: https://en.wikipedia.org/wiki/Weak_symbol.

Wellons C. Legitimate Use of Variable Length Arrays : blog. — URL: https://nullprogram.com/blog/2019/10/27/.

Werhausen S. Potential issue with C++20's initialization change // GitHub : site. — URL: https://gist.github.com/s9w/ad9b1dd1ea6fb17e956559c8b352e246.

What are some uses of decltype(auto)? // Stack Overflow : site. — URL: https://clck.ru/3JT2bw.

What are the rules about using an underscore in a C++ identifier? // Stack Overflow : site. — URL: https://clck.ru/3JSuZc.

What can I do with a moved-from object? // Stack Overflow : site. — URL: https://clck.ru/3JSNS6.

Why does flowing off the end of a non-void function without returning a value not produce a compiler error? // Stack Overflow : site. — URL: https://goo.su/TwtsidU.

Why isn't vector<bool> a STL container? // Stack Overflow : site. — URL: https://stackoverflow.com/questions/17794569/why-isnt-vectorbool-a-stl-container.

Wording for P2644R1 Fix for Range-based for Loop // ISO JTC1/SC22/WG21: Programming Language C++. — URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2718r0.html.

Working Draft Programming Languages — C++. Copy/move elision. — URL: https://eel.is/c++draft/class.copy.elision.

Working Draft Programming Languages — C++. Ranges library. — URL: https://eel.is/c++draft/range.req.

Writing efficient matrix product expressions // Eigen : site. — URL: https://eigen.tuxfamily.org/dox/TopicWritingEfficientProductExpression.html.

Yaghmour S. Exploring Undefined Behavior Using Constexpr : blog. — URL: https://clck.ru/3JRDyo.

Yaghmour S. The Usual Arithmetic Confusions : blog. — URL: https://shafik.github.io/c++/2021/12/30/usual_arithmetic_confusions.html.

Воробьёв Н. Как переменная может быть не равной ее собственному значению // Хабр : сайт. — URL: https://habr.com/ru/articles/307702/.

Гельвих М. Как не надо проверять размер массива в C++ // PV-Studio : сайт. — URL: https://pvs-studio.ru/ru/blog/posts/cpp/1112/.

Документация PVS-Studio. V1032. Pointer is cast to a more strictly aligned pointer t ype // PV-Studio : сайт. — URL: https://pvs-studio.ru/ru/docs/warnings/v1032/.

Документация PVS-Studio. V1036. Potentially unsafe double-checked locking // PV-Studio : сайт. — URL: https://pvs-studio.ru/ru/docs/warnings/v1036/.

Дубовик А. Ох уж этот std::make_shared... // Хабр : сайт. — URL: https://habr.com/ru/articles/509004/.

Зинин М. C++: сеанс спонтанной археологии и почему не стоит использовать вариативные функции в стиле C // Хабр : сайт. — URL: https://habr.com/ru/articles/430064/.

Карпов А. 60 антипаттернов для C++ программиста // PV-Studio : сайт. — URL: https://pvs-studio.ru/ru/blog/posts/cpp/1053/.

Карпов А. Анализатор кода не прав, да здравствует анализатор // PV-Studio : сайт. — URL: https://pvs-studio.ru/ru/blog/posts/cpp/0779/.

Карпов А. Изменения выравнивания типов и последствия // PV-Studio : сайт. — URL: https://pvs-studio.ru/ru/blog/posts/cpp/0009/.

Карпов А. Четыре причины проверять, что вернула функция malloc // PV-Studio : сайт. — URL: https://pvs-studio.ru/ru/blog/posts/cpp/0938/.

Мальцев М. std::launder: the most obscure new feature of C++17 : blog. — URL: https://github.com/miyuki.

Неручек Е. Продление жизни временных значений в C++: рецепты и подводные камни // PV-Studio : сайт. — URL: https://pvs-studio.ru/ru/blog/posts/cpp/1006/.

Проблема остановки // Рувики : сайт. — URL: https://ru.ruwiki.ru/wiki/Проблема_остановки.

Теорема Райса // Рувики : сайт. — URL: https://ru.ruwiki.ru/wiki/Теорема_Райса.