

РАЙНЕР ГРИММ

0 + + 20

В ДЕТАЛЯХ

Эта книга посвящена новому стандарту C++20. В ней вначале приводится краткий обзор C++, а далее рассматриваются ключевые возможности языка:

- концепты – семантические категории для параметров шаблона, позволяющие более точно выразить свои намерения относительно разрабатываемой программы с помощью средств языка программирования;
- модули – являются более эффективной заменой заголовочным файлам;
- новая библиотека диапазонов – позволяет выполнять алгоритмы непосредственно в контейнере, объединять их и применять к непрерывным потокам данных;
- корутины – основа асинхронного программирования в C++20. Они предназначены для программирования кооперативных задач, циклов событий, потоков данных и конвейеров.

Многочисленные примеры кода позволят вам изучить передовые возможности C++ и применять их на практике.



Райнер Гримм работает архитектором программного обеспечения с 1999 года. В 2002 он организовал в своей компании цикл повышения квалификации и с тех пор проводит учебные курсы. С 2016 года является независимым инструктором, ведет семинары по C++ и Python.

Гримм в силу своей профессии всегда ищет лучший способ преподавания C++ и на основе своего опыта опубликовал несколько книг о современном C++.

В свободное время пишет статьи о C++, Python и Haskell в собственном блоге, а также выступает на профильных конференциях.

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК «Галактика»
books@aliens-kniga.ru

DMK
ИЗДАТЕЛЬСТВО
www.dmk.pf

ISBN 978-5-97060-956-9



9 785970 609569 >

С++20 в деталях

Райнер Гримм



Москва, 2023

УДК 004.42

ББК 32.372

Г82

Главный научный редактор:

Романов А. Ю. – канд. техн. наук, доцент Московского института электроники и математики им. А. Н. Тихонова Национального исследовательского университета «Высшая школа экономики».

Райнер Гримм

Г82 C++20 в деталях / пер. с англ. А. В. Борескова; под науч. ред. А. Ю. Романова, И. И. Романовой. – М.: ДМК Пресс, 2023. – 518 с.: ил.

ISBN 978-5-97060-956-9

В этой книге подробно рассказывается о новом стандарте C++20. Для тех, кто незнаком с C++20, приводится его краткий обзор, а далее рассматриваются ключевые возможности языка. Вы получите представление о ключевых изменениях в ядре языка (концепты и модули), новой библиотеке диапазонов, корутинах, а затем сможете применить теорию на практике, изучив ряд примеров. Книгу можно использовать как справочное руководство и изучать главы в удобном для вас порядке.

Издание будет полезно разработчикам, желающим освоить последнюю версию C++, изучить передовые возможности и добавления в язык, а также заглянуть за кулисы разработки новых стандартов языка и узнать, как предлагаются, обсуждаются и утверждаются новые изменения в стандарт C++ и чем вызваны эти изменения.

Книга, которую вы держите в руках, открывает серию «Книжная полка Истового Инженера», которая издается при поддержке компании YADRO. Это издание подготовлено к публикации Московским институтом электроники и математики им. А. Н. Тихонова НИУ ВШЭ совместно с «ДМК Пресс».

УДК 004.42

ББК 32.372

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

ISBN (анг.) 979-8-73298-945-8

ISBN (рус.) 978-5-97060-956-9

© 2020 Rainer Grimm

© Оформление, издание, перевод, ДМК Пресс, 2023

© Научное редактирование, НИУ ВШЭ, 2023

Оглавление

https://t.me/it_boooks/2

Предисловие от издательства	11
Отзывы и пожелания.....	11
Список опечаток.....	11
Нарушение авторских прав	11
Предисловие от главного редактора русского перевода	12
Истории читателей.....	16
Введение	17
Соглашения	17
Специальные шрифты	17
Специальные блоки.....	18
Исходный код.....	18
Компиляция программ	18
Как вам следует читать эту книгу?	20
Личные замечания	20
Благодарности	20
Сиппи	20
Редакторы русского перевода.....	21
Обо мне	22
О ЯЗЫКЕ C++.....	24
1. Исторический контекст	25
1.1 C++98.....	25
1.2 C++03.....	26
1.3 TR1	26
1.4 C++11.....	26
1.5 C++14.....	26
1.6 C++17.....	26

2. Стандартизация	27
2.1 Стадия 3	28
2.2 Стадия 2	28
2.3 Стадия 1	28
КРАТКИЙ ОБЗОР C++20	30
3. C++20	31
3.1 Большая четверка	32
3.1.1 Концепты	32
3.1.2 Модули	33
3.1.3 Библиотека диапазонов	34
3.1.4 Корутины	35
3.2 Ядро языка	37
3.2.1 Оператор трехстороннего сравнения	37
3.2.2 Назначенная инициализация	38
3.2.3 <code>constexpr</code> и <code>constexpr</code>	40
3.2.4 Улучшения работы с шаблонами	41
3.2.5 Улучшения лямбд	42
3.2.6 Новые атрибуты	42
3.3 Стандартная библиотека	43
3.3.1 <code>std::span</code>	43
3.3.2 Улучшения контейнеров	44
3.3.3 Арифметические утилиты	44
3.3.4 Календарь и временные зоны	44
3.3.5 Библиотека для форматированного вывода	45
3.4 Параллельность	46
3.4.1 Атомарные операции	46
3.4.2 Семафоры	47
3.4.3 Защелки и барьеры	47
3.4.4 Кооперативное прерывание	48
3.4.5 <code>std::jthread</code>	50
3.4.6 Синхронные выходные потоки	51
ПОДРОБНО ПРО C++20	54
4. Ядро языка	55
4.1 Концепты	55
4.1.1 Два неправильных подхода	56
4.1.2 Преимущества концептов	62
4.1.3 Длинная, длинная история	62
4.1.4 Использование концептов	63
4.1.5 Ограниченные или неограниченные заполнители	75
4.1.6 Сокращенные шаблонные функции	78
4.1.7 Предопределенные концепты	82
4.1.8 Определение концептов	88

4.1.9 Применение концептов.....	96
4.2 Модули.....	108
4.2.1 Для чего нужны модули?.....	108
4.2.2 Преимущества использования модулей.....	114
4.2.3 Простой пример использования модулей.....	115
4.2.5 Экспорт из модуля.....	120
4.2.6 Рекомендации по структуре модуля.....	121
4.2.7 Блок интерфейса модуля и блок реализации модуля.....	122
4.2.8 Подмодули и разделы модулей.....	125
4.2.9 Шаблоны в модулях.....	129
4.2.10 Линковка на уровне модулей.....	132
4.2.11 Заголовочные блоки.....	134
4.3 Оператор трехстороннего сравнения.....	136
4.3.1 Упорядочение до C++20.....	136
4.3.2 Упорядочение начиная со стандарта C++20.....	138
4.3.3 Категории сравнения.....	141
4.3.4 Создаваемый компилятором оператор трехстороннего сравнения.....	143
4.3.5 Переписывание выражений.....	148
4.3.6 Задаваемые пользователем и создаваемые автоматически операторы сравнения.....	151
4.4 Назначенная инициализация.....	154
4.4.1 Агрегированная инициализация.....	154
4.4.2 Именованная инициализация членов класса.....	156
4.5 consteval и constexpr.....	161
4.5.1 consteval.....	161
4.5.2 constexpr.....	163
4.5.3 Выполнение функций.....	164
4.5.4 Инициализация переменных.....	166
4.5.5 Исправляем проблему порядка статической инициализации.....	167
4.6 Улучшение работы с шаблонами.....	173
4.6.1 Условный явный конструктор.....	173
4.6.2 Нетипизированные параметры шаблона.....	176
4.7 Улучшения лямбд.....	180
4.7.1 Шаблонные параметры для лямбд.....	180
4.7.2 Определение неявного копирования указателя this.....	184
4.7.3 Лямбды в контекстах без выполнения. Использование конструктора по умолчанию и копирования для лямбд без состояния.....	186
4.8 Новые атрибуты.....	190
4.8.1 [[nodiscard("reason")]].....	191
4.8.2 [[likely]] и [[unlikely]].....	195
4.8.3 [[no_unique_address]].....	196
4.9 Дополнительные улучшения.....	199
4.9.1 volatile.....	199
4.9.2 Оператор цикла for с инициализацией на основе диапазона.....	201
4.9.3 Виртуальная функция с constexpr.....	202

4.9.4 Новый символьный тип для utf8-строк: <code>char8_t</code>	204
4.9.5 Использование <code>using enum</code> в локальной области видимости	205
4.9.6 Инициализаторы по умолчанию для битовых полей	206

5. Стандартная библиотека209

5.1 Библиотека диапазонов	210
5.1.1 Концепты <code>ranges</code> и <code>views</code>	211
5.1.2 Работа алгоритмов непосредственно со всем контейнером	212
5.1.3 Композиция функций	216
5.1.4 Отложенное выполнение	218
5.1.5 Определение видов	221
5.1.6 Аромат Python	224
5.2 <code>std::span</code>	230
5.2.1 Статическая и динамическая длина	230
5.2.2 Автоматический вывод размера непрерывной последовательности объектов	232
5.2.3 Создание <code>std::span</code> из указателя и размера	233
5.2.4 Изменение объектов, к которым происходит обращение через ссылку	235
5.2.5 Обращение к элементам <code>std::span</code>	236
5.2.6 Постоянный диапазон изменяемых элементов	238
5.3 Улучшения контейнеров	241
5.3.1 Контейнеры и алгоритмы со спецификатором <code>constexpr</code>	241
5.3.2 <code>std::array</code>	242
5.3.3 Последовательное удаление из контейнеров	244
5.3.4 <code>contains</code> для ассоциативных контейнеров	249
5.3.5 Проверка строки на наличие префикса и суффикса	252
5.4 Арифметические функции	255
5.4.1 Безопасное сравнение целых чисел	255
5.4.2 Математические константы	260
5.4.3 Вычисление середины отрезка и линейная интерполяция	262
5.4.4 Работа с битами	263
5.5 Календарные зоны и часовые пояса	269
5.5.1 Время дня	270
5.5.2 Календарные даты	273
5.5.3 Часовые пояса	289
5.6 Библиотека форматирования	296
5.6.1 Функции форматирования	296
5.6.2 Форматная строка	298
5.6.3 Задаваемые пользователем типы	306
5.7 Дальнейшие улучшения	312
5.7.1 <code>std::bind_front</code>	312
5.7.2 <code>std::is_constant_evaluated</code>	314
5.7.3 <code>std::source_location</code>	316

6. Параллельность	318
6.1 Корутины	319
6.1.1 Функция-генератор	320
6.1.2 Характеристики	323
6.1.3 Фреймворк	325
6.1.4 Ожидаемые и ожидающие объекты	328
6.1.5 Исполняемый поток процессов	330
6.1.6 <code>co_return</code>	334
6.1.7 <code>co_yield</code>	336
6.1.8 <code>co_await</code>	339
6.2 Атомарные переменные	349
6.2.1 <code>std::atomic_ref</code>	349
6.2.2 Атомарный умный указатель	358
6.2.3 Расширения <code>std::atomic_flag</code>	362
6.2.4 Расширения <code>std::atomic</code>	370
6.3 Семафоры	374
6.4 Защелки и барьеры	379
6.4.1 <code>std::latch</code>	379
6.4.2 <code>std::barrier</code>	385
6.5 Координированное прерывание	389
6.5.1 <code>std::stop_source</code>	390
6.5.2 <code>std::stop_token</code>	391
6.5.3 <code>std::stop_callback</code>	391
6.6 <code>std::jthread</code>	398
6.6.1 Автоматическое присоединение	399
6.6.2 Кооперативное прерывание <code>std::jthread</code>	401
6.7 Синхронизированные потоки вывода	404
7. Практические примеры	413
7.1 Быстрая синхронизация потоков	414
7.1.1 Условные переменные	415
7.1.2 <code>std::atomic_flag</code>	417
7.1.3 <code>std::atomic<bool></code>	420
7.1.4 Семафоры	422
7.1.5 Общая статистика	424
7.2 Вариации объектов <code>future</code>	425
7.2.1 Ленивый объект <code>future</code>	428
7.2.2 Выполнение на другом потоке	431
7.3 Модификация и обобщение генератора	436
7.3.1 Изменения	440
7.3.2 Обобщение	443
7.4 Различные потоковые архитектуры, основанные на задачах	447
7.4.1 Прозрачная архитектура ожидающего потока задач	447
7.4.2 Автоматическое возобновление ожидающей задачи	450
7.4.3 Автоматическое возобновление ожидающего объекта на отдельном потоке	453

ЭПИЛОГ	458
ДОПОЛНИТЕЛЬНАЯ ИНФОРМАЦИЯ	460
8. C++23 и не только	461
8.1 C++23	462
8.1.1 Библиотека сопрограмм	462
8.1.2 Модуляризированная стандартная библиотека	476
8.1.3 Исполнители	479
8.1.4 Сетевая библиотека	483
8.2 C++23 или позже	485
8.2.1 Контракты	485
8.2.2 Рефлексия	488
8.2.3 Сопоставление с образцом	492
8.3 Дополнительная информация о стандарте C++23	496
9. Дополнительное тестирование	497
10. Глоссарий	510

Предисловие от издательства

Отзывы и пожелания

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

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

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

Список опечаток

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

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

Нарушение авторских прав

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

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

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

Предисловие от главного редактора русского перевода

Дорогие друзья!

Книга, которую вы держите в руках, открывает серию «Книжная полка Истового Инженера», которая издается при поддержке компании YADRO. Это издание подготовлено к публикации Московским институтом электроники и математики им. А. Н. Тихонова НИУ ВШЭ совместно с «ДМК Пресс».

Если вы интересуетесь цифровой электроникой, разработкой на ПЛИС, проектированием на языках описания аппаратуры Verilog или VHDL, то вы, скорее всего, уже знакомы с книгами серии «Цифровой синтез», такими как: Д. Харрис и С. Л. Харрис «Цифровая схемотехника и архитектура компьютера»; «Цифровой синтез: практический курс» (под. ред. А. Ю. Романова и Ю. В. Панчула), Ф. Бруно «Программирование FPGA для начинающих» и др. Эта книга несколько отличается по тематике и в первую очередь ориентирована на программистов, работающих с языками высокого уровня.

Почему я взялся редактировать русский перевод книги Р. Гримма «C++20 в деталях»?

На самом деле я сильно сомневался. Но меня убедила моя коллега Ирина Романова, она же и выполнила первую вычитку и редактуру этой книги. Без ее помощи я бы не согласился редактировать данный материал. Остальные мотивы описаны далее.

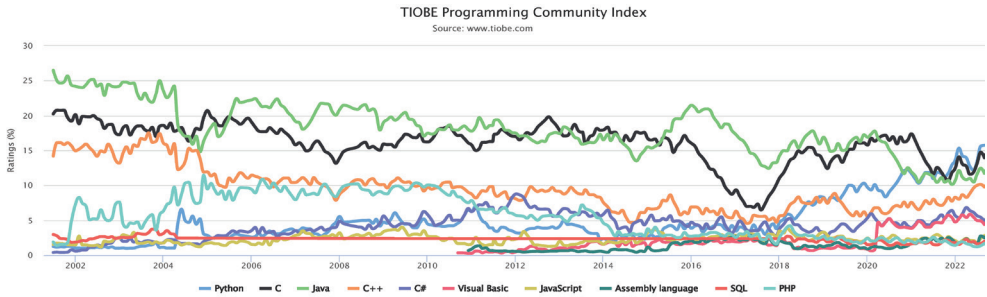
Как я использую C++?

Хотя я применяю C++ в своей практике постоянно, использую я его обычно как язык системного программирования, а также для программирования встраиваемых систем (таких как микроконтроллеры), т. е. вполне могу обходиться подмножеством C; а если нужна объектно-ориентированность, то вполне достаточно тех возможностей, которые предоставляет стандарт C++98.

Зачем изучать C++20?

Зачем же тогда изучать все эти C++11, потом C++14 и т. д., а теперь и C++20? (Ведь на подходе уже и следующий стандарт!)

Ответ: это очень интересно! C++ – это один из ведущих языков. По рейтингу ТIOB C/C++ – все еще самый популярный язык в мире. Популярнее, чем Python!



C++ – довольно «старый» язык, который развивается, меняется, создает новые методы и принципы программирования прямо у нас на глазах. Это (во многом) отражение прошлого и будущего всех классических высокоуровневых языков программирования. Каждый новый стандарт – это новые парадигмы и «фишки» программирования. При этом предлагают, описывают и внедряют их обычные программисты в открытом комьюнити.

Поэтому даже если вы заядлый «питонист» или никогда не планируете использовать новшества C++, все равно стоит делать как я: следить за выходом новых стандартов и хотя бы пролистывать релизы о нововведениях в язык и другую литературу по теме – потому что понимать ход современной передовой мысли разработчиков программного кода, знать новые приемы и возможности ключевого языка общения всех программистов очень важно для саморазвития любого, кто любит и практикует разработку программ.

Зачем нужен русский перевод этой книги?

Очень часто в среде программистов встречаю мнение, что изучать специализированную литературу по программированию надо на языке оригинала: английском. В целом я с этим согласен – сам большинство информации по теме получаю на английском. Но это не потому, что читать на английском удобней, а потому, что на русском почти всегда нужной информации нет. Если хотите читать оригинал, пожалуйста, читайте его. Но если кто-то хочет быстро познаться с новшествами C++20 или разобраться более детально с какими-то тонкостями этих новшеств на примерах – эта книга для вас.

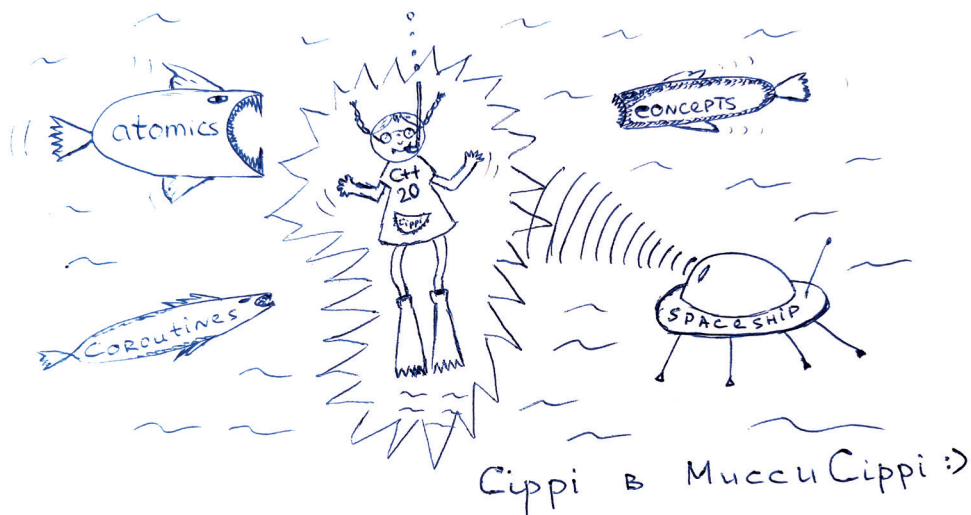
Почему именно книга Р. Гримма?

Потому что это признанный автор, а его книги очень популярны. Потому что Р. Гримм сам из «тусовки» разработчиков новых стандартов C++, и у него сравнительно легкая и интересная подача. Надеюсь, нам удалось это сохранить и в русском переводе. А еще в этой книге довольно информативные и показательные примеры с небольшим количеством ошибок. Этим могут похвастаться далеко не все издания в данной области, даже издания именитых авторов!

Еще для себя я выработал один неформальный принцип оценки зарубежных книг, рассчитанных на большую аудиторию: это то, как она оформлена. Чем больше вложено в оформление издания, тем выше вероятность того (хоть и не всегда!), что содержание книги будет на уровне. Дело в том, что западное

общество довольно инфантильно (чего только стоят «График Шму» или «Принцип печенья Орео»!), поэтому книге без «картинок» довольно сложно добиться популярности. Объединение «картинок» и хорошего содержания является фактором, обеспечивающим успех книги.

Данная книга соответствует этому критерию. Хотя, на мой взгляд, персонаж книги, Сиппи, довольно безобразна, а подписи к рисункам катастрофически неинформативны. Такое представление персонажа я (не художник) тоже выдать могу! Но не включаю его в серьезную книгу. Хотя... а почему бы и нет?



Было желание убрать все эти бессмысленные рисунки из книги, но, в конце концов, было решено смириться с этим.

Для кого эта книга?

Давайте вернемся к серьезному тону и, наконец, ответим на вопрос: кому будет интересна эта книга? Программирование C++ по ней вы не изучите – для этого есть классические книги Х. Дейтела, Б. Страуструпа, С. Прата и др. Для того чтобы воспринимать материал, нужно уже знать концепцию объектно-ориентированного программирования и иметь хотя бы небольшой опыт разработки программ на C++ или «близких по духу» высокоуровневых языках – вроде C#, Java, Python. Неплохо бы знать такие концепции, как лямбда-функции и многопоточные/параллельные программы. Стандарт C++20 глубоко вы по этой книге тоже не изучите. Она нужна для первого быстрого и легкого знакомства с новшествами, которые появились в новом стандарте языка. Заодно читатели смогут узнать немного о причинах появления этих новшеств, принципах и идеях, которые в них заложены. Также в книге очень неплохие и понятные примеры программ. Остальные тонкости вы при необходимости узнаете на практике или изучив стандарт.

Ошибки, терминология и другие замечания

Проблема перевода такого рода книг состоит в том, что довольно часто в русском языке нет устоявшегося аналога английскому термину или прямой русский перевод выглядит глупо и неестественно. В основном это так, потому что сами английские термины являются «новоделами». Например, тот же coroutine. Все уже смирились говорить и писать «корутин», притом что в русском языке есть тоже иностранное, но более привычное слово «сопрограмма», которое имеет приблизительно то же значение. В русском переводе осуществлена попытка соблюсти баланс, обеспечивающий читабельность и понятность текста: какие-то термины используются в форме, более привычной русскоязычному читателю, а какие-то вообще не переведены. По той же причине сознательно не переведены надписи на рисунках и комментарии в листингах.

Эту книгу прочитали два научных редактора, один корректор и еще два студента (огромная благодарность Александру Богомолову и Руслану Нуржанову), но, безусловно, она не идеальна. Я буду очень признателен тем внимательным читателям, которые обнаружат в данном издании какие-либо ошибки или опечатки и сообщат о них по адресу a.romanov@hse.ru или dmkpress@gmail.com (книги постоянно перепечатываются, и в каждом новом тираже все найденные ошибки и недочеты исправляются).

Также вы можете присылать свои рисунки, изображающие Сиппи. Я пока еще не придумал, что с ними делать, но (возможно) получится импровизированная выставка 😊

Истории читателей



Сандор Дарго, старший программист в Amadeus

«С++ 20 в деталях» – это именно та книга, которая вам сейчас нужна, если вы хотите погрузиться в последнюю версию С++. Это полное руководство, в котором Рейнер рассматривает не просто передовые возможности С++20, но и небольшие добавления в язык. Эта книга содержит множество примеров кода, так что даже если у вас еще нет доступа к последним компиляторам, у вас все равно будет хорошее представление о том, чего вы можете ожидать от различных возможностей языка. Крайне рекомендую прочитать.



Адриан Там, директор по Data Science, Synectron Inc

С++ очень сильно развился с момента его появления на свет. С++20 – это практически новый язык. Конечно, это книга не предназначена для того, чтобы научить вас наследованию или перегрузке, но если вы хотите привести свои знания С++ к современному уровню, то это очень подходящая книга. Вы удивитесь тому, сколько всего нового из С++20 вошло в базовый язык С++. Эта книга дает ясные объяснения и краткие примеры. Структура книги позволит вам использовать ее как справочник в дальнейшем. Она поможет вам узнать все современные возможности языка.

Введение

Моя книга по C++ – это и учебник, и справочное руководство. Она научит вас C++20 и даст вам детальную информацию о новом стандарте C++. Наиболее выдающимися возможностями C++20 являются следующие:

- **концепты** – изменяют способ, которым мы думаем о программах с шаблонами. Они являются семантическими категориями для параметров шаблонов. Они позволяют вам ясно выражать свои намерения через систему типов. Если что-то пойдет не так, то компилятор даст вам понятное сообщение об ошибке;
- **модули** – позволяют обойти ограничения заголовочных файлов. Они очень многообещающи. Например, разделение заголовочного файла и исходного файла становится таким уже устаревшим, как и препроцессор. В результате вы получаете более быструю компиляцию и легкий способ создания пакетов;
- новая **библиотека диапазонов** (ranges library) поддерживает применение алгоритмов к контейнерам, включающим алгоритмы, организованные как конвейер (pipe), и «ленивое» (lazy) применение алгоритмов к бесконечным потокам данных;
- благодаря **корутинам** (coroutines) асинхронное программирование в C++ становится широко распространенным. Корутины образуют основу для кооперативных задач, циклов обработки событий, бесконечных потоков данных или конвейеров.

Конечно, это далеко не все. Вот еще некоторые новые возможности C++20:

- генерируемые автоматически операторы сравнения;
- библиотеки для работы с датами и временными зонами;
- библиотека format;
- «виды» (view) непрерывных блоков памяти;
- улучшенные прерываемые потоки;
- атомарные умные указатели;
- семафоры;
- примитивы управления, такие как защелки (latch) и барьеры (barrier).

Соглашения

Есть всего несколько соглашений по форматированию этой книги:

Специальные шрифты

Курсив

Используется *курсив* для выделения цитат.

Жирный

Используется **жирный** шрифт для выделения имен.

Моноширинный

Используется моноширинный шрифт для кода, инструкций, ключевых слов и имен типов, переменных, функций и классов.

Специальные блоки

Блоки содержат подсказки, предупреждения и дополнительную информацию.



Подсказка

Этот блок предоставляет дополнительную информацию по текущему материалу и подсказки по компиляции программ.



Предупреждения

Блоки предупреждений должны помочь вам избегать ошибок.



Собранная информация

Этот блок в конце каждого раздела содержит важную информацию, которую стоит запомнить.

Исходный код

Примеры исходного кода, приведенные в книге, являются завершенными. Это значит, что если у вас есть подходящий компилятор, то вы можете их откомпилировать и выполнить. Имя исходного файла находится в заголовке каждого примера кода. Исходный код использует четыре пробела для табуляции (отступов). Иногда используется два пробела из соображений размещения на странице.

Кроме того, я не большой любитель директив namespace вроде `using namespace std`, поскольку они делают код более сложным для чтения и могут засорять пространства имен. Соответственно, я применяю их только тогда, когда это улучшает читаемость кода (например, `using namespace std::chrono_literals`). Иногда для более удобного размещения кода на странице я использую `using`, например `using std::chrono::system_clock`.

Таким образом, только в случае необходимости иногда делаются следующие отступления от общих правил оформления кода:

- табуляция на два пробела, а не на четыре;
- используется директива `using namespace std`.

Компиляция программ

Поскольку C++20 – это новый стандарт, то многие примеры могут быть откомпилированы только при помощи подходящих компиляторов. Я использую последние версии GCC¹, Clang² и MSVC³. При компиляции программы вы должны указать используемый стандарт C++. Это значит, что при использовании GCC

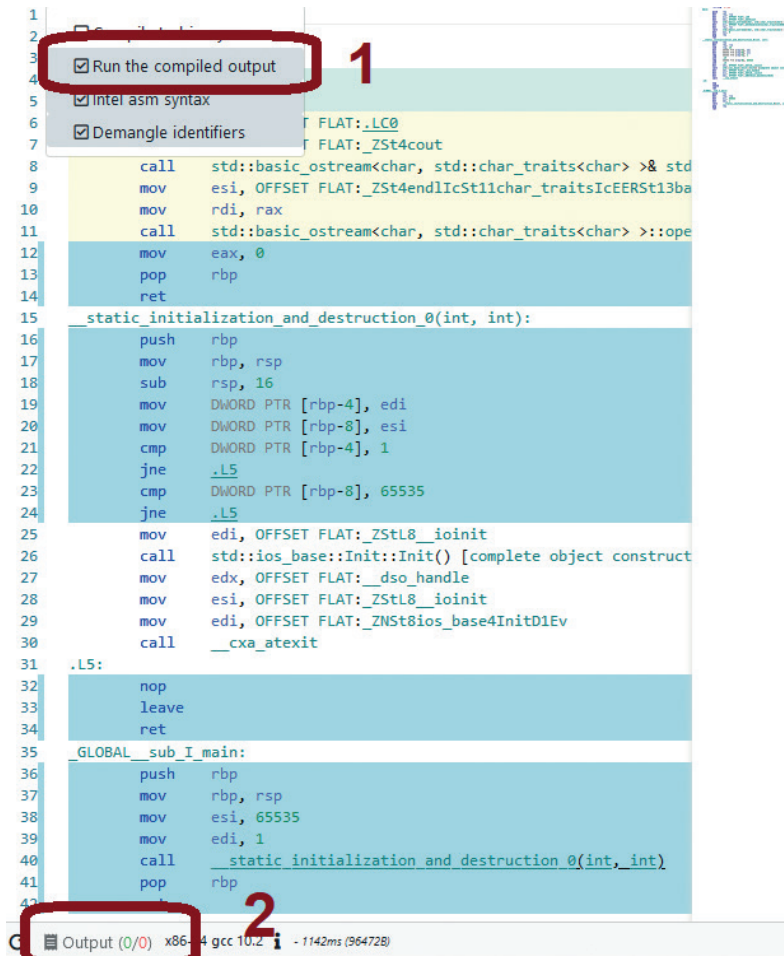
¹ <https://gcc.gnu.org>.

² <https://clang.llvm.org>.

³ https://en.wikipedia.org/Microsoft_Visual_C%2B%2B.

или Clang вы должны указать флаг `-std=c++20` и для MSVC флаг `/std:c++latest`. При применении параллельности (concurrency) для GCC и Clang необходимо указать, что вам требуется линковка с библиотекой `pthread` при помощи опции `-pthread`.

Если у вас нет в распоряжении подходящего компилятора C++, то вы можете использовать онлайн-компилятор вроде Wandbox¹ или Compiler Explorer². При применении Compiler Explorer с выбором компилятора GCC или Clang вы также можете выполнить откомпилированную программу. Для этого, во-первых, вам нужно выбрать **Run the compiled output** и, во-вторых, открыть окно **Output**.



Run code in the Compiler Explorer

Вы можете получить дополнительную информацию о совместимости с C++20 для различных компиляторов на [cppreference.com](https://en.cppreference.com/w/cpp/compiler_support)³.

¹ <https://wandbox.org>.

² <https://godbolt.org>.

³ https://en.cppreference.com/w/cpp/compiler_support.

Как вам следует читать эту книгу?

Если вы незнакомы со стандартом C++20, то начните с краткого обзора для получения общей картины.

После получения общей картины вы можете переходить к ключевым возможностям языка (core language). Рассказ о каждой новой возможности языка является самодостаточным, но наилучшим вариантом будет прочтение всей книги от начала до конца. При первом прочтении вы можете пропустить то, что не упомянуто в кратком обзоре.

Личные замечания

Благодарности

Я просил читателей проверить на ошибки мой англоязычный блог Moderness-Cpp¹ и получил гораздо больше откликов, чем ожидал. Огромное спасибо вам всем. Вот имена тех, кто внес в это свой вклад: Bob Bird, Nicola Bombace, Dave Burchill, Sandor Dargo, James Drobina, Frank Grimm, Kilian Henneberger, Ivan «espkk» Kondakov, Péter Kardos, Rakesh Mane, Jonathan O'Connor, John Plaice, Iwan Smith, Paul Targosz, Steve Vinoski и Greg Wagner.

Особенное спасибо моей дочери Джульетте и моей жене Беатрис. Джульетта поправила мой язык и исправила множество опечаток. Беатрис создала персонажа Сиппи (Cippi, девочка на рисунках) и иллюстрации к этой книге.

Сиппи



Позвольте представить вам Сиппи, которая будет сопровождать вас в этой книге. Надеюсь, она вам понравится.

Я Сиппи (по аналогии с Пеппи Длинныйчулок), любознательная, умная и женственная!

¹ <http://modernescpp.com>.

Редакторы русского перевода



Романов Александр Юрьевич (<https://www.hse.ru/staff/a.romanov>) – главный научный редактор русского перевода данной книги, доцент Московского института электроники и математики им. А. Н. Тихонова Национального исследовательского университета «Высшая школа экономики» (МИЭМ НИУ ВШЭ). С 2014 г. работает в МИЭМ НИУ ВШЭ, где возглавляет лабораторию САПР (<https://miem.hse.ru/edu/ce/cadsystem>), специализирующуюся на проектной деятельности, а также разработке цифровых систем на ПЛИС/микроконтроллерах, робототехнических комплексов, аппаратных реализаций систем искусственного интеллекта, многопроцессорных систем, систем удаленного доступа к ла-

бораторному оборудованию и т. д. В 2015 г. защитил диссертацию в Институте проблем проектирования в микроэлектронике РАН (г. Зеленоград), является автором более 150 научных статей, патентов и книг. А. Ю. Романов преподает C++ с 2009 г. в качестве лекционного и практического курса для студентов и постоянно использует данный язык в практической работе.



Романова Ирина Ивановна (<https://www.hse.ru/staff/iromanova>) – научный редактор русского перевода – занимается преподаванием компьютерных и инженерных дисциплин с 2010 г. В настоящее время работает старшим преподавателем в Московском институте электроники и математики им. А. Н. Тихонова Национального исследовательского университета «Высшая школа экономики» (МИЭМ НИУ ВШЭ), ведущий преподаватель и лектор дисциплины «Информатика» для студентов 1-го курса, а также соавтор известной книги «Цифровой синтез: практический курс» (М.: ДМК Пресс, 2020). Автор более 30 научных статей. Специализируется на методике преподавания компьютерных дисциплин студентам младших курсов.

Обо мне

Я работал в качестве архитектора программных систем, руководителя группы (team lead) и лектора начиная с 1999-го. В 2002 году я создавал тренинги для обучения. Я начал вести занятия с 2002 года. Моими первыми тренингами были занятия по управлению проприетарным программным обеспечением, но вскоре я начал обучать языкам Python и C++. В мое свободное время мне нравится писать статьи о C++, Python и Haskell. Также мне нравится выступать на различных конференциях. Каждую неделю я пишу что-то в моем англоязычном блоге ModernesCpp и в блоге на немецком языке¹, размещенном у Heise Developer.

С 2016 года я выступал в качестве независимого инструктора с семинарами по современному C++ и Python. Я написал несколько книг на разных языках о современном C++ и, в частности, по параллелизму. В связи с моей профессией я всегда ищу лучшие пути обучения современному C++.



Райнер Гримм

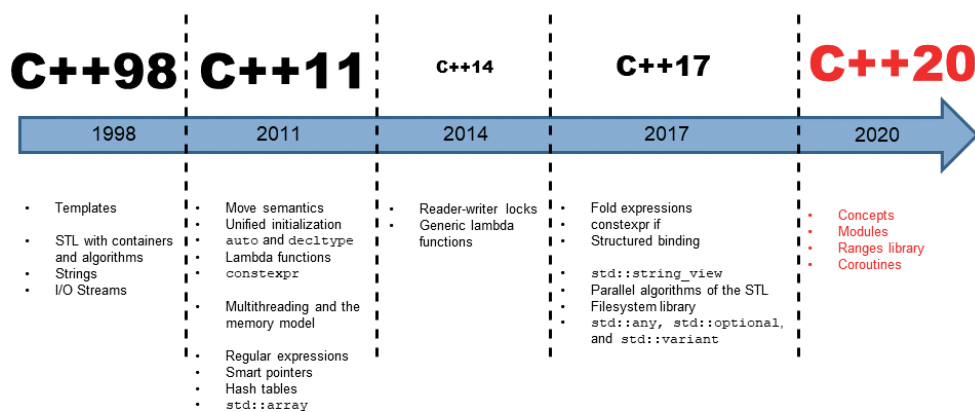
¹ <https://www.grimm-jaud.de/index.php/blog>.

О ЯЗЫКЕ C++

1. Исторический контекст

https://t.me/it_boooks/2

C++20 – это следующий большой шаг после стандарта C++11. Как и C++11, C++20 меняет способ, как мы программируем, используя современный C++. Эти изменения произошли в основном вследствие добавления в язык таких понятий, как концепты (Concepts), модули (Modules), диапазоны (Ranges) и корутины (Coroutines). Для понимания данного шага в эволюции C++ позвольте мне сказать несколько слов об историческом контексте C++20.



История C++

1.1 C++98

В конце 80-х Бьярн Страуструп и Маргарет А. Эллис написали свою знаменитую книгу *Annotated C++ Reference*¹ (ARM). Данная книга выполняла две функции – определяла функциональность C++ в мире, в котором существуют различные реализации C++, и стала основой для первого стандарта C++98 (ISO/IEC 14882). В число важных возможностей языка вошли шаблоны и стандартная библиотека шаблонов (Standard Template Library, STL) со своими контейнерами, алгоритмами и строками, а также потоки ввода/вывода.

¹ <https://stroustrup.com/arm.html>.

1.2 C++03

Со стандартом C++03 (14882:2003) C++98 получил небольшое уточнение, настолько небольшое, что ему даже не выделено места на таймлайне. В сообществе C++03, включающий C++98, обычно называется **legacy C++**.

1.3 TR1

В 2005 году произошло очень важное событие. Был опубликован документ под названием Technical Report 1. TR1 был огромным шагом к C++11 и, соответственно, современному C++. TR1 (TR 19768) был основан на проекте Boost¹, который был создан членами комитета по стандартизации C++. TR1 содержит 13 библиотек, которые должны были стать частью C++11. В их число входят библиотека регулярных выражений, библиотека для работы со случайными числами, умные указатели и хеш-таблицы. А вот специальным математическим функциям пришлось ждать до C++17.

1.4 C++11

Мы называем C++11 *современным C++*. Это же название *современный C++* используется и для C++14 и C++17. C++11 внес много новых возможностей, которые принципиально изменили наше программирование на C++. Например, в C++11 вошли добавления из TR1, а также семантика перемещения (move semantics), форвардинг (perfect forwarding), вариадические шаблоны (variadic templates) и constexpr. Но это еще не все. С появлением стандарта C++11 мы также получили, причем в первый раз, модель памяти как основу работы с нитями и стандартизированный API для работы с нитями.

1.5 C++14

C++14 – это довольно небольшой стандарт. Он привнес read-write блокировки, обобщенные лямбда-функции и расширенные constexpr-функции.

1.6 C++17

C++17 не является ни большим, ни маленьким. Он привнес две выдающиеся возможности: параллельный STL и стандартный API для работы с файловой системой. Порядка 80 алгоритмов из стандартной библиотеки шаблонов могут выполняться параллельно или быть векторизованы. Как и с C++11, библиотеки boost оказали огромное влияние на C++17. Библиотеки boost предоставили библиотеку для работы с файловой системой и новые типы: `std::string_view`, `std::optional`, `std::variant` и `std::any`.

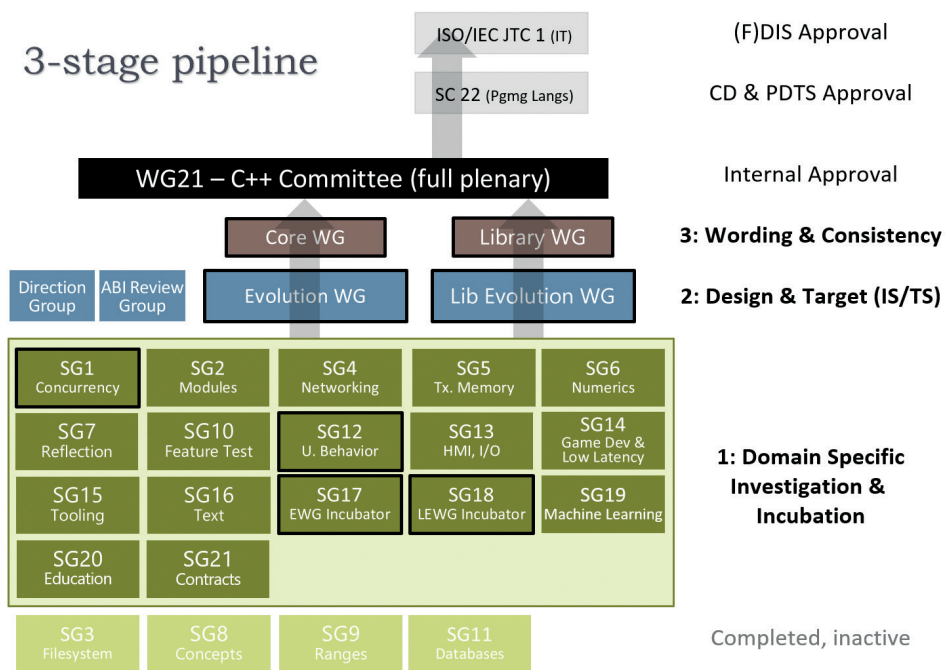
¹ <https://www.boost.org>.

2. Стандартизация

Процесс стандартизации C++ довольно демократичен. Комитет по стандартизации языка называется WG21 (Working Group 21) и был сформирован в 1990–1991 годах. Подкомитетами являются:

- организатор (Convener): возглавляет WG21, назначает график встреч и группы изучения;
- редактор проекта: применяет изменения к черновику стандарта C++;
- секретарь: назначает встречи WG21.

3-stage pipeline



Группы стандартизации C++

Комитет организован в конвейер из трех стадий, состоящий из нескольких подгрупп.

2.1 Стадия 3

Стадия 3 предназначена для уточнения формулировок и обеспечения непротиворечивости предлагаемых добавлений и состоит из двух групп: формулировок по самому языку (core language wording, CWG) и формулировок по библиотекам (library wording).

2.2 Стадия 2

Стадия 2 состоит из двух групп: развитие самого языка (Core Language Evolution, EWG) и развитие библиотек (Library Evolution, LWEG). EWG и LWEG отвечают за новые возможности, включающие в себя расширения языка и стандартных библиотек соответственно.

2.3 Стадия 1

Стадия 1 предназначена для изучения направлений развития языка. Члены групп встречаются лицом к лицу, между встречами по телефону и видеоконференциями. Центральные группы могут изучать результаты целевых групп для обеспечения согласованности и последовательности изменений языка.

Ниже приводится список целевых групп (Study Groups):

- **SG1, Concurrency** – параллельные вычисления, включая модель памяти;
- **SG2, Modules** – темы, связанные с модулями;
- **SG3, File system** – файловая система;
- **SG4, Networking** – развитие сетевой библиотеки;
- **SG5, Transactional memory** – транзакционная память для включения в будущие релизы;
- **SG6, Numerics** – численные расчеты, числа с фиксированной точкой, числа с плавающей точкой и дроби;
- **SG7, Compile time programming** – программирование в контексте компиляции программы;
- **SG8, Concepts** – концепты;
- **SG9, Ranges** – диапазоны;
- **SG10, Feature test** – переносимые тесты для проверки того, поддерживает ли конкретная реализация конкретную возможность (feature);
- **SG11, Databases** – интерфейсы для взаимодействия с базами данных;
- **SG12, UB & Vulnerabilities** – улучшения, направленные борьбу с уязвимостями и неопределенным/незаданным поведением текущей версии стандарта языка;
- **SG13 HMI & I/O (Human/Machine Interface)** – поддержка устройств ввода/вывода;
- **SG14, Game development & low latency** – поддержка требования работы с небольшой задержкой;
- **SG15, Tooling** – инструменты для разработчиков, включая модули и пакеты;
- **SG16, Unicode** – обработка на C++ текста на юникоде;
- **SG17, EWG Incubator** – ранние обсуждения развития языка;

- **SG18, LWEG Incubator** – ранние обсуждения развития библиотек;
- **SG19, Machine Learning** – темы, связанные с искусственным интеллектом, и линейная алгебра;
- **SG20, Education** – материалы для обучающих курсов по C++;
- **SG21, Contracts** – поддержка языком контрактного проектирования (Design by contract);
- **SG22 C/C++ Liason** – обсуждения взаимодействия C и C++.

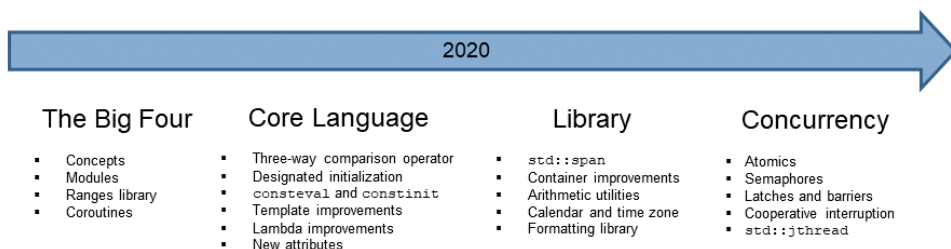
Этот раздел дал вам краткий обзор стандартизации C++ и структуры комитета по стандартизации C++. Вы можете получить дополнительную информацию на сайте: <https://isocpp.org/std>.

КРАТКИЙ ОБЗОР C++20

3. C++20

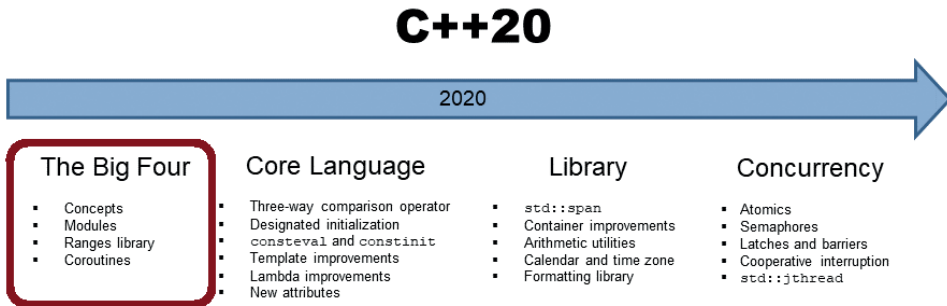
Прежде чем погружаться в детали C++20, я хотел бы дать краткий обзор возможностей C++20. Этот обзор служит сразу двум целям: дать первое впечатление и предоставить ссылки на соответствующие разделы, которые вы можете использовать для погружения в детали. Также в этой главе будут только фрагменты кода, а не завершённые программы.

C++20



В C++20 вошло несколько выдающихся возможностей: концепты, диапазоны, корутины и модули. Каждая заслуживает своего раздела.

3.1 Большая четверка



Каждая из новых возможностей языка меняет то, как мы программируем на современном C++. Давайте начнем с концептов.

3.1.1 Концепты

Обобщенное программирование с шаблонами позволяет определять функции и классы, которые могут быть использованы с различными типами данных. В результате бывает, что мы инстанцируем шаблон с неверным типом. В итоге мы можем получить много страниц непонятных ошибок. Эта проблема решается при помощи концептов. Концепты дают вам возможность описать требования для параметров шаблонов, которые будут проверяться компилятором, и тем самым улучшают то, как мы думаем и пишем обобщенный код. Это достигается благодаря тому, что:

- требования для параметров шаблона становятся частью публичного интерфейса шаблона;
- перегрузка функций или специализация классов может быть основана на концептах;
- мы получаем улучшенные сообщения об ошибках, поскольку компилятор проверяет требования к параметрам шаблона для переданных параметров.

Кроме того, у концептов есть еще преимущества:

- вы можете использовать уже готовые концепты для написания собственных;
- унифицируется использование ключевого слова `auto` и концептов. Вместо `auto` вы теперь можете использовать концепт;
- если объявление функции использует концепт, то эта функция автоматически становится шаблонной функцией. Написание шаблонных функций становится таким же легким, как и написание обычных функций.

Следующий фрагмент кода показывает определение и использование концепта `Integral`.

Определение и использование концепта `Integral`

```
template <typename T>
concept Integral = std::is_integral<T>::value;
```

```
Integral auto gcd(Integral auto a, Integral auto b) {
    if( b == 0 ) return a;
    else return gcd(b, a % b);
}
```

Концепт `Integral` требует от своего параметра-типа `T`, чтобы `std::is_integral<T>::value` было истинным. Оно, в свою очередь, является частью библиотеки трейтов (type traits), проверяющей во время компиляции, что данный тип является целым. Если `std::is_integral<T>::value` истинно, то все в порядке, иначе вы получите сообщение об ошибке.

Алгоритм `gcd` находит наибольший общий делитель, используя алгоритм Евклида¹. Этот фрагмент кода использует так называемый сокращенный синтаксис для функции-шаблона для определения `gcd`. Функция `gcd` требует, чтобы оба ее аргумента и выходной тип поддерживали концепт `Integral`. Другими словами, `gcd` – это что-то вроде шаблонной функции, задающей требования на входные аргументы и возвращаемый тип. Когда я уберу весь «синтаксический сахар», вы увидите природу этой функции.

Использование концепта `Integral` в опции `requires`

```
template<typename T>
requires Integral<T>
T gcd(T a, T b) {
    if( b == 0 ) return a;
    else return gcd(b, a % b);
}
```

Описатель `requires` задает требования на тип параметров `gcd`.

3.1.2 Модули

Модули обеспечивают:

- более быструю компиляцию;
- уменьшение необходимости использования макросов;
- более выраженную логическую структуру кода;
- делают устаревшими заголовочные файлы;
- помощь в избавлении от уродливых макрорешений.

Вот пример простого модуля `math`:

¹ <https://en.wikipedia.org/wiki/Euclid>.

Модуль `math`

```
1  export module math;
2
3  export int add(int fir, int sec) {
4      return fir + sec;
5  }
```

Выражение `export module math` (строка 1) является объявлением модуля. Поместив `export` перед описанием функции `add` (строка 3), мы экспортируем эту функцию. Теперь она может быть использована потребителем данного модуля.

Использование модуля `math`

```
import math;

int main() {

    add(2000, 20);

}
```

Выражение `import math` импортирует модуль `math` и делает экспортируемые имена видимыми в текущей области видимости.

3.1.3 Библиотека диапазонов

Библиотека диапазонов (`range`) поддерживает алгоритмы, которые могут:

- работать напрямую с контейнерами, не требуя итераторов для задания диапазона;
- выполняться отложено (`lazy`);
- совмещаться/комбинироваться при помощи символа `|`.

Говоря проще: эта библиотека поддерживает функциональные шаблоны.

Следующий пример демонстрирует композицию функций через символ `|`.

Композиция функций при помощи символа `|`

```
1  int main() {
2      std::vector<int> ints{0, 1, 2, 3, 4, 5};
3      auto even = [](int i){ return i % 2 == 0; };
4      auto square = [](int i) { return i * i; };
5
6      for (int i : ints | std::views::filter(even) |
7                  std::views::transform(square)) {
8          std::cout << i << ' ';           // 0 4 16
9      }
10 }
```

Лямбда-выражение для `even` (строка 3) – это лямбда, возвращающая `true`, если ее аргумент четный. Лямбда `square` (строка 4) переводит свой аргумент в его квадрат. Строки 6 и 7 показывают композицию функций, которую нужно читать слева направо: `for (int i : ints | std::views::filter(even) | std::views::transform (square))`. К каждому элементу `ints` применяет фильтр `even`, и к оставшимся элементам применяется преобразование `square`. Если вы знакомы с функциональным программированием, то это совсем не ново.

3.1.4 Корутины

Корутины – это обобщенные функции, выполнение которых можно приостанавливать и возобновлять потом, сохраняя их состояние. Корутины очень удобны для написания событийно-ориентированных (event-driven) приложений. Такими приложениями могут быть симуляции, игры, серверы, пользовательские интерфейсы и даже алгоритмы. Корутины обычно используются для кооперативной многозадачности.

C++20 не предоставляет конкретных корутинов, вместо этого C++20 предоставляет фреймворк для написания корутинов. Этот фреймворк состоит из более чем 20 функций, некоторые из которых вы должны реализовать, некоторые можете переопределить. Таким образом можно приспособить корутины для своих целей.

Следующий фрагмент кода использует генератор для создания потенциально бесконечного потока данных. В главе «Корутины» содержится реализация `Generator`.

Генератор для получения бесконечного потока данных

```

1  Generator<int> getNext(int start = 0, int step = 1){
2      auto value = start;
3      while (true) {
4          co_yield value;
5          value += step;
6      }
7  }
8
9  int main() {
10
11      std::cout << '\n';
12
13      std::cout << "getNext():";
14      auto gen1 = getNext();
15      for (int i = 0; i <= 10; ++i) {
16          gen1.next();
17          std::cout << " " << gen1.getValue();
18      }
19

```

```
20     std::cout << "\n\n";
21
22     std::cout << "getNext(100, -10):";
23     auto gen2 = getNext(100, -10);
24     for (int i = 0; i <= 20; ++i) {
25         gen2.next();
26         std::cout << " " << gen2.getValue();
27     }
28
29     std::cout << "\n";
30
31 }
```

Функция `getNext` является корутином, поскольку она использует ключевое слово `co_yield`. Есть бесконечный цикл, который возвращает значение через `co_yield` (строка 4). После возвращения из вызова `getNext` корутин приостанавливается до следующего вызова. Есть один тонкий момент: возвращаемое значение – это `Generator<int>`. Здесь начинаются хитрости, описанные в разделе по корутинам.

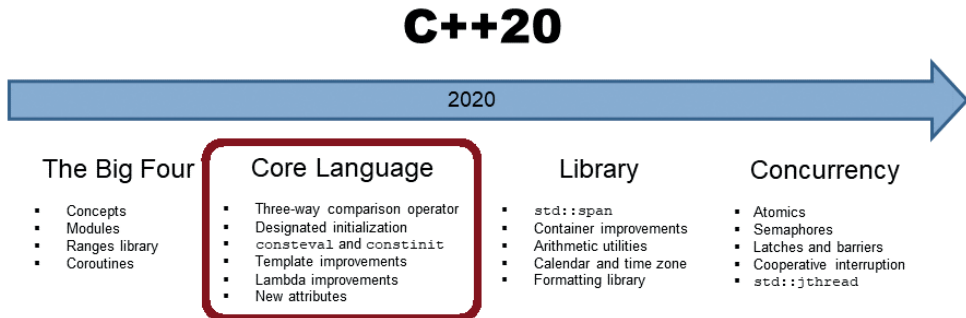
```
Start

getNext(): 0 1 2 3 4 5 6 7 8 9 10

getNext(100, -10): 100 90 80 70 60 50 40 30 20 10 0 -10 -20 -30 -40 -50 -60 -70 -80 -90 -100
0
Finish
```

Бесконечный генератор данных

3.2 Ядро языка



3.2.1 Оператор трехстороннего сравнения

Оператор трехстороннего сравнения (spaceship operator) `<=>` сравнивает два значения – A и B – и определяет, что имеет место `A < B`, `A == B` или `A > B`.

Если вы определили оператор трехстороннего сравнения как `default`, то компилятор попытается сгенерировать соответствующий оператор для вашего класса. В этом случае вы также получите все шесть операторов сравнения: `==`, `!=`, `<`, `<=`, `>` и `>=`.

Автоматическое создание трехстороннего оператора сравнения

```
struct MyInt {
    int value;
    MyInt(int value): value{value} { }
    auto operator<=>(const MyInt&) const = default;
};
```

Созданный компилятором оператор `<=>` выполняет лексикографическое сравнение, начиная с базовых классов и используя все нестатические члены класса в порядке их описания. Чтобы более подробно разобраться с данным оператором, рекомендую довольно хитрый пример из блога Microsoft: [Simplify Your Code with Rocket Science: C++20's Spaceship operator](https://devblogs.microsoft.com/cppblog/simplify-your-code-with-rocket-science-c20s-spaceship-operator/)¹.

¹ <https://devblogs.microsoft.com/cppblog/simplify-your-code-with-rocket-science-c20s-spaceship-operator/>.

3.2.2 Назначенная инициализация

Оператор трехстороннего сравнения для производных классов

```
struct Basics {
    int i;
    char c;
    float f;
    double d;
    auto operator<=>(const Basics&) const = default;
};

struct Arrays {
    int ai[1];
    char ac[2];
    float af[3];
    double ad[2][2];
    auto operator<=>(const Arrays&) const = default;
};

struct Bases : Basics, Arrays {
    auto operator<=>(const Bases&) const = default;
};

int main() {
    constexpr Basics a = { { 0, 'c', 1.f, 1. },
        { { 1 }, { 'a', 'b' }, { 1.f, 2.f, 3.f }, { { 1., 2. }, { 3., 4. } } } };
    constexpr Basics b = { { 0, 'c', 1.f, 1. },
        { { 1 }, { 'a', 'b' }, { 1.f, 2.f, 3.f }, { { 1., 2. }, { 3., 4. } } } };
    static_assert(a == b);
    static_assert(!(a != b));
    static_assert(!(a < b));
    static_assert(a <= b);
    static_assert(!(a > b));
    static_assert(a >= b);
}
```

Я полагаю, что наиболее сложным в этом фрагменте кода является не оператор трехстороннего сравнения, а инициализация класса Base. Агрегированная инициализация (aggregate initialization) означает, что вы можете непосредственно проинициализировать члены класса (class, struct или union), если все они public. В этом случае вы можете использовать список инициализации в фигурных скобках, как в приведенном примере.

Прежде чем я приступлю к назначенной инициализации (designated initialization), давайте еще немного рассмотрим агрегатную инициализацию. Вот простой пример.

Агрегатная инициализация

```
struct Point2D{
    int x;
    int y;
};

class Point3D{
public:
    int x;
    int y;
    int z;
};

int main(){

    std::cout << "\n";

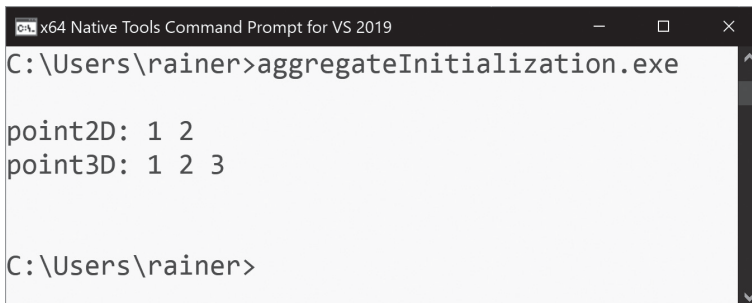
    Point2D point2D {1, 2};
    Point3D point3D {1, 2, 3};

    std::cout << "point2D: " << point2D.x << " " << point2D.y << "\n";
    std::cout << "point3D: " << point3D.x << " "
                << point3D.y << " " << point3D.z << "\n";

    std::cout << '\n';

}
```

Ниже приводится вывод данного примера.



```
x64 Native Tools Command Prompt for VS 2019
C:\Users\rainer>aggregateInitialization.exe

point2D: 1 2
point3D: 1 2 3

C:\Users\rainer>
```

Агрегатная инициализация очень чувствительная к ошибкам, поскольку вы можете поменять местами аргументы конструктора и никогда этого не заметить. Явная инициализация лучше неявной. Давайте посмотрим, что это значит. Взглянем на назначенную инициализацию из C99, которая стала частью стандарта C++.

Назначенная инициализация

```
1  struct Point2D{
2      int x;
3      int y;
4  };
5
6  class Point3D{
7  public:
8      int x;
9      int y;
10     int z;
11 };
12
13 int main(){
14
15     Point2D point2D { .x = 1, .y = 2 };
16     // Point2D point2d { .y = 2, .x = 1 };           // error
17     Point3D point3D { .x = 1, .y = 2, .z = 2 };
18     // Point3D point3D { .x = 1, .z = 2 }           // {1, 0, 2}
19
20
21     std::cout << "point2D: " << point2D.x << " " << point2D.y << "\n";
22     std::cout << "point3D: " << point3D.x << " " << point3D.y << " " << point3D.z
23         << "\n";
24
25 }
```

Аргументы для экземпляров Point2D и Point3D явно именуются. Вывод этой программы идентичен выводу предыдущей. Закомментированные строки 16 и 18 довольно интересны. Строка 16 дает ошибку, потому что порядок инициализаторов не соответствует порядку, в котором описаны члены класса. Что касается строки 18, то здесь пропущен инициализатор для y. В этом случае y будет проинициализирован нулем, так же как и при списке инициализации { 1, 0, 3}.

3.2.3 consteval и constexpr

Новый спецификатор `constexpr`, добавленный в C++20, создает непосредственную функцию (immediate function). Для такой функции каждый ее вызов должен происходить за константное время компиляции. Непосредственная функция – это просто `constexpr`-функция, но обратное не всегда верно.

Непосредственная функция

```
constexpr int sqr(int n) {
    return n*n;
}

constexpr int r = sqr(100);  // OK

int x = 100;
int r2 = sqr(x);              // Error
```

Последнее присваивание приводит к ошибке, поскольку x – это не константное выражение, и поэтому вычисление `sqr(x)` не может быть выполнено на этапе компиляции.

Опция `constinit` гарантирует, что статические переменные или TLS-переменные будут проинициализированы во время компиляции. Статические переменные – это переменные, память для которых выделяется при запуске программы. У TLS-переменных время жизни привязано ко времени жизни потока.

`constinit` гарантирует, что эти переменные будут проинициализированы во время компиляции. Но это не означает их константности.

3.2.4 Улучшения работы с шаблонами

C++20 предоставляет многочисленные улучшения для программирования шаблонов. Общий конструктор (generic constructor) – это универсальный конструктор, поскольку вы можете вызвать его с любым типом.

Неявный и явный общие конструкторы

```
struct Implicit {
    template <typename T>
    Implicit(T t) {
        std::cout << t << '\n';
    }
};

struct Explicit {
    template <typename T>
    explicit Explicit(T t) {
        std::cout << t << '\n';
    }
};

Explicit exp1 = "implicit";  // Error
Explicit exp2{"explicit"};
```

Общий конструктор класса `Implicit` слишком общий. Помещая ключевое слово `explicit` перед конструктором, как у класса `Explicit`, конструктор становится явным. Это значит, что неявные преобразования больше не валидны.

3.2.5 Улучшения лямбд

В C++20 лямбды получили много новых улучшений. У них могут быть шаблонные параметры, невычисляемый контекст и лямбды без собственного состояния могут быть созданы конструктором по умолчанию (`default constructor`) и конструктором копирования (`copy constructor`). Более того, компилятор может определять, когда вы неявно копируете указатель `this`, – это значит, заметная часть **неопределенного поведения** (`undefined behavior`) у лямбд ушла в прошлое.

Если вы хотите определить лямбду, которая принимает только `std::vector`, то шаблонные параметры для лямбд позволяют это сделать.

Шаблонные параметры для лямбды

```
auto foo = [<typename T>(std::vector<T> const& vec) {  
    // do vector-specific stuff  
};
```

3.2.6 Новые атрибуты

В C++20 появились новые атрибуты, включая `[[likely]]` и `[[unlikely]]`. Оба этих атрибута позволяют дать подсказку оптимизатору, задавая, какой путь выполнения будет чаще или реже встречаться.

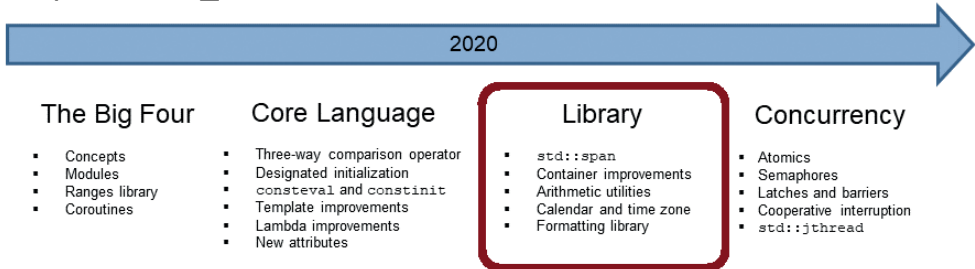
Атрибут `[[likely]]`

```
for(size_t i=0; i < v.size(); ++i){  
    if (v[i] < 0) [[likely]] sum -= sqrt(-v[i]);  
    else sum += sqrt(v[i]);  
}
```

3.3 Стандартная библиотека

https://t.me/it_boooks/2

C++20



3.3.1 `std::span`

Класс `std::span` представляет собой объект, который может ссылаться на непрерывную последовательность объектов. Иногда `std::span` называют видом (view), но он никогда не является владельцем (owner). Этим видом может быть стандартный массив языка C, `std::array`, указатель с размером или `std::vector`. Типичная реализация `std::span` требует указатель на первый элемент массива и его размер. Основной причиной появления `std::span` является то, что простой массив превращается в указатель при передаче в функцию и его размер теряется. `std::span` автоматически рассчитывает размер массива, `std::array` или `std::vector`. Если вы используете указатель для инициализации `std::span`, то вам в конструкторе необходимо передать размер.

`std::span` как аргумент функции

```
void copy_n(const int* src, int* des, int n){}
```

```
void copy(std::span<const int> src, std::span<int> des){}
```

```
int main(){

    int arr1[] = {1, 2, 3};
    int arr2[] = {3, 4, 5};

    copy_n(arr1, arr2, 3);
    copy(arr1, arr2);

}
```

По сравнению с функцией `copy_n`, функция `copy` не требует передачи количества элементов. Поэтому с появлением `std::span<T>` уходит одна из распространенных причин ошибок в программах.

3.3.2 Улучшения контейнеров

В C++20 появилось много улучшений, относящихся к контейнерам из стандартной библиотеки шаблонов (Standard Template Library). Прежде всего у `std::vector` и `std::string` появились конструкторы `constexpr`, которые могут вызываться на этапе компиляции. Контейнеры из стандартной библиотеки поддерживают последовательное стирание контейнеров (`consistent container erasure`), а ассоциативные контейнеры поддерживают функцию `contains`. Также `std::string` позволяет проверять префикс или суффикс строки.

3.3.3 Арифметические утилиты

Сравнение знакового и беззнакового целых чисел является довольно хитрым случаем неопределенного поведения и ошибок. Благодаря новым безопасным функциям для сравнения целых, `std::cmp_*`, этот источник коварных ошибок ушел в прошлое.

Безопасное сравнение целых

```
int x = -3;
unsigned int y = 7;

if (x < y) std::cout << "expected";
else std::cout << "not expected";           // not expected

if (std::cmp_less(x, y)) std::cout << "expected"; // expected
else std::cout << "not expected";
```

Кроме того, C++20 включает в себя целый ряд математических констант, в том числе e , π или ϕ в пространстве имен `std::numbers`.

Новые возможности по работе с отдельными битами дают прямой доступ к отдельным битам и последовательностям битов.

Доступ к отдельным битам и последовательностям битов

```
std::uint8_t num= 0b10110010;

std::cout << std::has_single_bit(num) << '\n';           // false
std::cout << std::bit_width(unsigned(5)) << '\n';       // 3
std::cout << std::bitset<8>(std::rotl(num, 2)) << '\n'; // 11001010
std::cout << std::bitset<8>(std::rotr(num, 2)) << '\n'; // 10101100
```

3.3.4 Календарь и временные зоны

Библиотека `chrono` из C++11 была дополнена новой функциональностью для работы с календарными типами и временными зонами. В календарные типы входят год, месяц, день недели и день месяца. Можно комбинировать эти базовые типы для получения составных сложных типов, например `year_month`, `year_month_day`, `year_month_day_last`, `year_month_weekday` и `year_month_weekday_last`. Оператор «/» был перегружен для удобного задания положения во времени. Также были добавлены новые литералы – “d” для дня и “y” для года.

Положения во времени могут быть показаны с использованием различных временных зон. В связи с расширением библиотеки `chrono` теперь можно легко реализовать следующие случаи:

- представление дат в заданных форматах;
- получение последнего дня месяца;
- получение количества дней между двумя датами;
- печать текущего времени в различных временных зонах (часовых поясах).

Следующая программа выводит локальное время в различных часовых поясах.

Локальное время в различных часовых поясах

```
using namespace std::chrono;

auto time = floor<milliseconds>(system_clock::now());
auto localTime = zoned_time<milliseconds>(current_zone(), time);
auto berlinTime = zoned_time<milliseconds>("Europe/Berlin", time);
auto newYorkTime = zoned_time<milliseconds>("America/New_York", time);
auto tokyoTime = zoned_time<milliseconds>("Asia/Tokyo", time);

std::cout << time << '\n';           // 2020-05-23 19:07:20.290
std::cout << localTime << '\n';      // 2020-05-23 21:07:20.290 CEST
std::cout << berlinTime << '\n';     // 2020-05-23 21:07:20.290 CEST
std::cout << newYorkTime << '\n';    // 2020-05-23 15:07:20.290 EDT
std::cout << tokyoTime << '\n';      // 2020-05-24 04:07:20.290 JST
```

3.3.5 Библиотека для форматированного вывода

Новая библиотека для форматированного вывода предоставляет безопасную и расширяемую альтернативу использованию `printf`. Она предназначена для дополнения существующих потоков ввода/вывода (I/O streams) и переиспользования части соответствующей инфраструктуры, такой как перегруженные операторы вставки для задаваемых пользователем типов.

```
std::string message = std::format("The answer is {}", 42);
```

Библиотека `std::format` использует для форматирования синтаксис из языка Python. Следующий пример показывает несколько стандартных примеров использования этой библиотеки.

- Форматирование и использование позиционных аргументов:

```
std::string s = std::format("I'd rather be {1} than {0}.", "right", "happy");
// s == "I'd rather be happy than right."
```

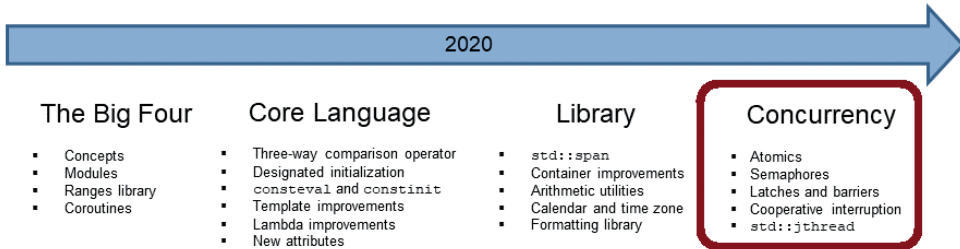
- Превращение целочисленной переменной в строку безопасным способом:

```
memory_buffer buf;
std::format_to(buf, "{}", 42); // replaces itoa(42, buffer, 10)
std::format_to(buf, "{:x}", 42); // replaces itoa(42, buffer, 16)
```

- Форматирование заданных пользователем типов.

3.4 Параллельность

C++20



3.4.1 Атомарные операции

Шаблон `std::atomic_ref` применяет атомарные операции (атомики, `atomics`) для доступа к неатомарному объекту. В результате одновременное чтение и запись не будут приводить к условию гонки (`race condition`). Время жизни объекта, к которому происходит обращение, должно превышать время жизни `std::atomic_ref`. Обращение к подобъекту объекта с `std::atomic_ref` не является потокобезопасным.

В соответствии с `std::atomic`¹, `std::atomic_ref` может быть специализирована и поддерживает специализацию для встроенных типов.

```
struct Counter {
    int a;
    int b;
};
```

```
Counter counter;
```

```
std::atomic_ref<Counter> cnt(counter);
```

В стандарте C++20 имеется два атомарных умных указателя, являющихся специализациями `std::atomic`: `std::atomic<std::shared_ptr<T>>` и `std::atomic<std::weak_ptr<T>>`. Оба атомарных умных указателя гарантируют, что не только управляющий блок, как в случае `std::shared_ptr`², но и сам связанный объект являются потокобезопасными.

Также новые расширения получил и `std::atomic`. C++20 содержит специализации для чисел с плавающей точкой. Это очень удобно, когда вы инкрементируете значение с плавающей точкой параллельно.

Значение типа `std::atomic_flag`³ является своего рода атомарным логическим значением. Оно может быть сброшено (`clear`) и переведено в установ-

¹ <https://en.cppreference.com/w/cpp/atomic/atomic>.

² https://en.cppreference.com/w/cpp/memory/shared_ptr.

³ https://en.cppreference.com/w/cpp/atomic/atomic_flag.

ленное (set) состояние. Для простоты я буду называть сброшенное состояние (clear state) false и установленное состояние (set state) true. Метод `clear()` позволяет установить его значение в false. Метод `test_and_set()` устанавливает значение в true и возвращает предыдущее значение. Метод для возвращения текущего значения ранее отсутствовал, и это изменилось в C++20, поскольку в `std::atomic_flag` появился метод `test()`.

Кроме того, `std::atomic_flag` может быть использован для синхронизации потоков при помощи функции `notify_one()`, `notify_all()` и `wait()`. В C++20 эти методы стали доступны для всех частичных и полных специализаций `std::atomic` и `std::atomic_ref`. Специализации доступны для логических типов, целочисленных типов, чисел с плавающей точкой и указателей.

3.4.2 Семафоры

Семафоры – это механизм синхронизации, используемый для одновременного доступа к совместно применяемому ресурсу. Семафор со счетчиком, как тот, который был добавлен в C++20, – это специальный тип семафора, у которого счетчик изначально больше нуля. Этот счетчик инициализируется в конструкторе. Получение (acquiring) семафора уменьшает счетчик, а его освобождение – увеличивает счетчик. Если поток пытается получить семафор, когда его счетчик равен нулю, то поток блокируется до тех пор, пока другой поток не освободит семафор.

3.4.3 Защелки и барьеры

Защелки (latches) и барьеры (barriers) – это простые примитивы синхронизации, которые позволяют некоторым нитям блокироваться до тех пор, пока счетчик не станет равным нулю. В чем разница между этими двумя механизмами синхронизации потоков? Вы можете использовать `std::latch` лишь один раз, а можете использовать `std::barrier` многократно. `std::latch` удобен для координации одной задачи группой потоков, а `std::barrier` удобен для управления повторяющимися задачами группой потоков. Кроме того, `std::barrier` может корректировать свой счетчик на каждой итерации.

Следующий пример кода основан на фрагменте из предложения N4204¹. Я всего лишь поправил несколько опечаток и переформатировал код.

Синхронизация потоков при помощи `std::latch`

```

1  void DoWork(threadpool* pool) {
2
3      std::latch completion_latch(NTASKS);
4      for (int i = 0; i < NTASKS; ++i) {
5          pool->add_task([&] {
6              // perform work
7              ...

```

¹ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4204.html>.

```
8         completion_latch.count_down();
9     });
10 }
11 // Block until work is done
12 completion_latch.wait();
13 }
```

Счетчик `std::latch completion_latch` изначально устанавливается в NTASKS (строка 3). После этого пул потоков выполняет NTASKS заданий (job) (строки 4–10). В конце каждого задания счетчик уменьшается на единицу (строка 8). Поток, выполняющий функцию `DoWork`, ожидает в строке 12 завершения всех заданий.

3.4.4 Кооперативное прерывание

При помощи `std::stop_token` выполнение `std::jthread` может быть прервано.

Прерывание выполнения `std::jthread`

```
1  int main() {
2
3      std::cout << '\n';
4
5      std::jthread nonInterruptible([]{
6          int counter{0};
7          while (counter < 10){
8              std::this_thread::sleep_for(0.2s);
9              std::cerr << "nonInterruptible: " << counter << '\n';
10             ++counter;
11         }
12     });
13
14     std::jthread interruptible([](std::stop_token stoken){
15         int counter{0};
16         while (counter < 10){
17             std::this_thread::sleep_for(0.2s);
18             if (stoken.stop_requested()) return;
19             std::cerr << "interruptible: " << counter << '\n';
20             ++counter;
21         }
22     });
23 }
```

```

24  std::this_thread::sleep_for(1s);
25
26  std::cerr << '\n';
27  std::cerr << "Main thread interrupts both jthreads" << std::endl;
28  nonInterruptible.request_stop();
29  interruptible.request_stop();
30
31  std::cout << '\n';
32
33  }

```

Функция `main` запускает два потока `nonInterruptible` и `interruptible` (строки 5 и 14). Только поток `interruptible` получает на вход `std::stop`, который в строке 18 используется для того, чтобы узнать, было ли запрошено прерывание потока. В случае прерывания лямбда незамедлительно завершает выполнение. Вызов `interruptible.request_stop()` запускает прерывание потока. Вызов `nonInterruptible.request_stop()` не влияет на работу соответствующего потока.

```

C:\Users\seminar>interruptJthread.exe

nonInterruptible: 0
interruptible: 0
nonInterruptible: 1
interruptible: 1
nonInterruptible: 2
interruptible: 2
nonInterruptible: 3
interruptible: 3

Main thread interrupts both jthreads

nonInterruptible: 4
nonInterruptible: 5
nonInterruptible: 6
nonInterruptible: 7
nonInterruptible: 8
nonInterruptible: 9

C:\Users\seminar>

```

Прерывание выполнения потока

3.4.5 std::jthread

Имя `std::jthread` обозначает присоединяемый поток (joinable thread). Класс `std::jthread` расширяет класс `std::thread`¹, автоматически присоединяя запущенный поток. `std::jthread` может быть тоже прерван.

Класс `std::jthread` был добавлен в C++20 из-за неинтуитивного поведения `std::thread`. Если `std::thread` по-прежнему присоединяемый, то `std::terminate`² вызывается в его деструкторе. Поток `thr` является присоединяемым, если не были вызваны ни `thr.join()`, ни `std::detach()`.

Поток `thr` по-прежнему присоединяем

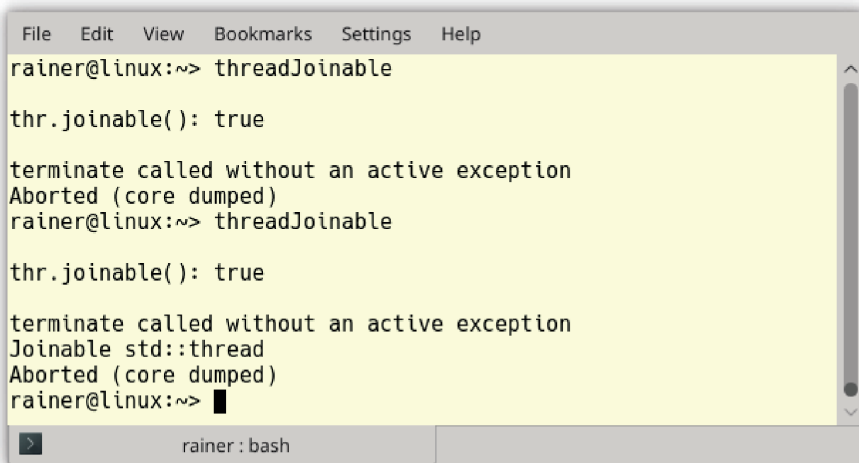
```
int main() {

    std::cout << '\n';

    std::cout << std::boolalpha;
    std::thread thr{[] { std::cout << "Joinable std::thread" << '\n'; }};
    std::cout << "thr.joinable(): " << thr.joinable() << '\n';

    std::cout << '\n';

}
```



```
File Edit View Bookmarks Settings Help
rainer@linux:~> threadJoinable
thr.joinable(): true
terminate called without an active exception
Aborted (core dumped)
rainer@linux:~> threadJoinable
thr.joinable(): true
terminate called without an active exception
Joinable std::thread
Aborted (core dumped)
rainer@linux:~> █
rainer: bash
```

Вызов `std::terminate` из присоединяемого потока

В обоих случаях выполнение программы прерывается. Но во втором у потока `thr` было достаточно времени для вывода сообщения "joinable std::thread".

¹ <https://en.cppreference.com/w/cpp/thread/thread>.

² <https://en.cppreference.com/w/cpp/error/terminate>.

В измененном примере я использую `std::jthread` из стандарта C++20.

Поток `thr` присоединяется автоматически

```
int main() {

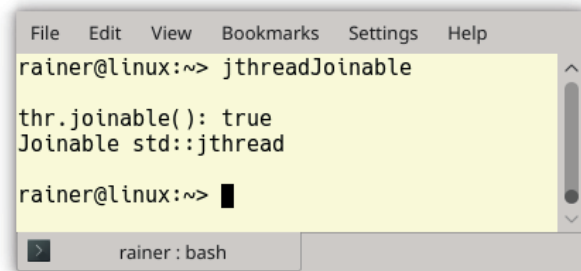
    std::cout << '\n';

    std::cout << std::boolalpha;
    std::jthread thr{[] { std::cout << "Joinable std::jthread" << '\n'; }};
    std::cout << "thr.joinable(): " << thr.joinable() << '\n';

    std::cout << '\n';

}
```

Теперь поток `thr` автоматически присоединяется в своем деструкторе, если это необходимо.



```
File Edit View Bookmarks Settings Help
rainer@linux:~> jthreadJoinable

thr.joinable(): true
Joinable std::jthread

rainer@linux:~> █
```

Поток `thr` присоединяется автоматически

3.4.6 Синхронные выходные потоки

В C++20 были добавлены синхронные выходные потоки (synchronized outputstreams). Что произойдет, когда несколько потоков одновременно пишут в `std::cout` без синхронизации?

Несинхронизированный вывод в `std::cout`

```
void sayHello(std::string name) {
    std::cout << "Hello from " << name << '\n';
}

int main() {

    std::cout << "\n";
```

```
std::jthread t1(sayHello, "t1");
std::jthread t2(sayHello, "t2");
std::jthread t3(sayHello, "t3");
std::jthread t4(sayHello, "t4");
std::jthread t5(sayHello, "t5");
std::jthread t6(sayHello, "t6");
std::jthread t7(sayHello, "t7");
std::jthread t8(sayHello, "t8");
std::jthread t9(sayHello, "t9");
std::jthread t10(sayHello, "t10");
```

```
std::cout << '\n';
```

```
}
```

В результате вы можете получить полную чепуху.

```
Hello from Hello from t1t2
Hello from t7
Hello from t8
Hello from t9
Hello from t3
Hello from t4
Hello from t5
Hello from Hello from t10t6
```

Несинхронизированный вывод в `std::cout`

Если мы перейдем к использованию `std::osyncstream(std::cout)` вместо `std::cout`, хаос превращается в гармонию.

Синхронизированный вывод в `std::cout`

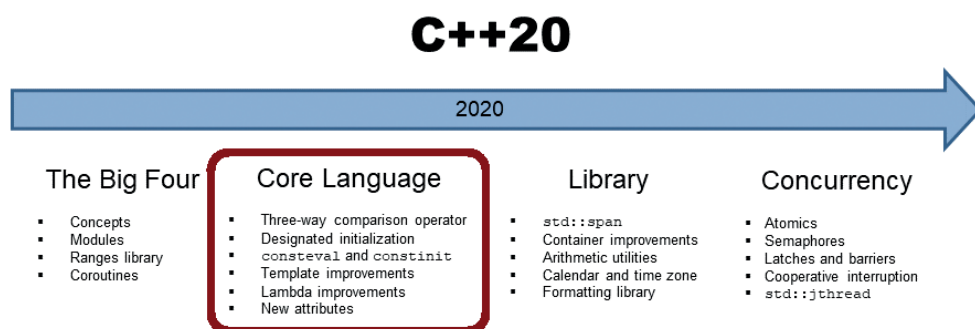
```
void sayHello(std::string name) {
    std::osyncstream(std::cout) << "Hello from " << name << '\n';
}
```

```
Hello from t1  
Hello from t2  
Hello from t3  
Hello from t4  
Hello from t5  
Hello from t6  
Hello from t7  
Hello from t8  
Hello from t9  
Hello from t10
```

Синхронизированный вывод в `std::cout`

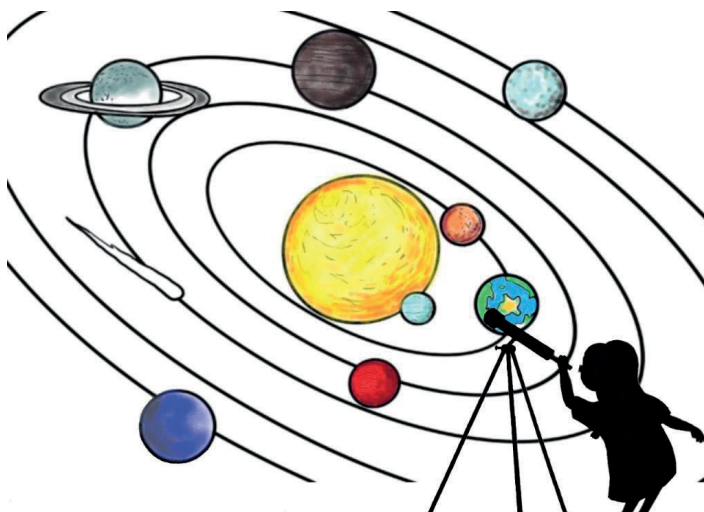
ПОДРОБНО ПРО C++20

4. Ядро языка



Концепты являются очень важной возможностью языка C++20. Поэтому они являются идеальной начальной точкой для рассказа о ключевых возможностях языка C++20.

4.1 Концепты



Сиппи изучает звезды

Для того чтобы объяснить важность концептов, я хочу начать с причины их появления.

4.1.1 Два неправильных подхода

До C++20 у нас было два диаметрально противоположных пути представления функций или методов – задание их для конкретных типов или задание их для общих (generic) типов. В последнем случае мы называем их шаблонными функциями и шаблонными классами. Но оба подхода имеют свой набор проблем.

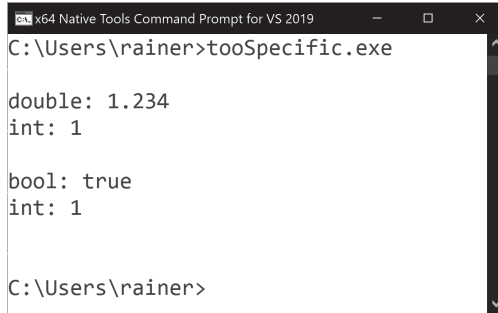
4.1.1.1 Слишком специализированный код

Сделать реализацию функции или класса для каждого типа может быть очень громоздкой работой. Чтобы избежать этого, часто на помощь приходят преобразования типов. Но то, что кажется спасением, нередко оказывается проклятием.

Неявные преобразования

```
1  // tooSpecific.cpp
2
3  #include <iostream>
4
5  void needInt(int i){
6      std::cout << "int: " << i << '\n';
7  }
8
9  int main(){
10
11      std::cout << std::boolalpha << '\n';
12
13      double d{1.234};
14      std::cout << "double: " << d << '\n';
15      needInt(d);
16
17      std::cout << '\n';
18
19      bool b{true};
20      std::cout << "bool: " << b << '\n';
21      needInt(b);
22
23      std::cout << '\n';
24
25  }
```

В первом случае (строка 13) я начинаю с `double` и заканчиваю `int` (строка 15). Во втором случае я начинаю с `bool` (строка 19) и заканчиваю `int` (строка 21).



```

C:\Users\rainer>tooSpecific.exe

double: 1.234
int: 1

bool: true
int: 1

C:\Users\rainer>

```

Неявные преобразования

Данная программа является примером двух неявных преобразований.

4.1.1.1.1 Сужающее преобразование

Вызывая `getInt(int a)` с аргументом типа `double`, получаем сужающее преобразование (narrowing conversion). Сужающее преобразование – это преобразование, включающее в себя потерю точности. Я полагаю, что это не то, что вы хотите.

4.1.1.1.2 Целочисленное расширение

Но другой вариант тоже не лучше. Вызывая `getInt(int a)` с параметром типа `bool`, мы переводим `bool` в `int`. Удивлены? Многие разработчики на C++ не знают, какой тип они получат, когда складывают две переменные типа `bool`.

Сложение двух `bool`

```

template <typename T>
auto add(T first, T second){
    return first + second;
}

int main(){
    add(true, false);
}

```

C++ Insights¹ показывает исходный код после инстанцииции шаблонной функции компилятором.

¹ <https://cppinsights.io/s/9bd14f99>.

```
1 template <typename T>
2 auto add(T first, T second){
3     return first + second;
4 }
5
6 #ifdef INSIGHTS_USE_TEMPLATE
7 template<>
8 int add<bool>(bool first, bool second)
9 {
10     return static_cast<int>(first) + static_cast<int>(second);
11 }
12 #endif
13
14
15 int main()
16 {
17     add(true, false);
18 }
```

Расширение bool в int

Строки 6–12 являются наиболее важными на этом изображении результатов¹. Инстанциация шаблонной функции add создает ее специализацию для возвращаемого типа int. Обе переменные типа bool неявно расширяются до int.

Я уверен в том, что мы полагаемся на магию преобразований типов для удобства, поскольку не хотим писать реализацию для каждого отдельного типа.

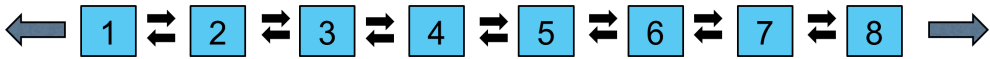
Давайте пойдем другим путем и используем обобщенную функцию. Может быть, это поможет?

4.1.1.2 Слишком обобщенный код

Давайте рассмотрим сортировку содержимого контейнера. Она должна работать для любого контейнера, чьи элементы можно упорядочивать. В следующем примере я применяю стандартный алгоритм `std::sort` к стандартному контейнеру `std::list`.

¹ <https://cppinsights.io/>.

Здесь `std::sort` использует странные типы аргументов `RandomIt`. На самом деле `RandomIt` обозначает итератор с произвольным доступом, и это дает нам ключ к пониманию сообщения об ошибке. Контейнер `std::list` предоставляет только двухсторонний итератор, но `std::sort` требует итератор с произвольным доступом. Следующий рисунок показывает, почему `std::list` не поддерживает итераторы с произвольным доступом.



Структура `std::list`

Если вы изучите документацию `std::sort` на cppreference.com, то вы найдете нечто очень важное – требования к шаблонным типам. Подобные требования к типам в C++20 помещаются в новую сущность – концепты.

4.1.1.3 На помощь приходят концепты

Концепты добавляют семантические требования к параметрам шаблона. У `std::sort` есть перегруженная версия, которая поддерживает оператор сравнения.

```
template< class RandomIt, class Compare >
constexpr void sort(RandomIt first, RandomIt last, Compare comp);
```

Ниже приводятся требования к самому продвинутому варианту `std::sort`.

- `RandomIt` должен удовлетворять `ValueSwappable` и `LegacyRandomAccessIterator`.
- Тип, получаемый при разыменовании `RandomIt`, должен удовлетворять `MoveAssignable` и `MoveConstructible`.
- Тип, получаемый при разыменовании `RandomIt`, должен удовлетворять `Compare`.

Требования, такие как `ValueSwappable` или `LegacyRandomAccessIterator`, являются именованными требованиями. Некоторые из этих требований формализованы в концептах C++20¹.

В частности, `std::sort` требует `LegacyRandomAccessIterator`. Давайте внимательно посмотрим на именованное требование `LegacyRandomAccessIterator`, называемое `random_access_iterator` (часть `<iterator>`) в C++20:

`std::random_access_iterator`

```
template<class I>
concept random_access_iterator =
    bidirectional_iterator<I> &&
    derived_from<ITER_CONCEPT(I), random_access_iterator_tag> &&
    totally_ordered<I> &&
    sized_sentinel_for<I, I> &&
```

¹ <https://en.cppreference.com/w/cpp/language/constraints>.


```
requires(I i, const I j, const iter_difference_t<I> n) {
    { i += n } -> same_as<I&>;
    { j + n } -> same_as<I>;
    { n + j } -> same_as<I>;
    { i -= n } -> same_as<I&>;
    { j - n } -> same_as<I>;
    { j[n] } -> same_as<iter_reference_t<I>>;
};
```

Тип `I` поддерживает концепт `random_access_iterator`, если он поддерживает концепт `bidirectional_iterator` и все из перечисленных требований. Например, требование `{ i += n } -> same_as<I&>` как часть выражения `requires` значит, что для значения типа `I` выражение `{ I += n }` валидно и возвращает значение типа `I&`. Чтобы завершить историю с сортировкой, `std::list` поддерживает `bidirectional_iterator`, но не `random_access_iterator`, как требует `std::sort`.

Теперь, когда вы используете алгоритм, который требует `random_access_iterator`, но поддерживается только `bidirectional_iterator`, вы получите четкое и понятное сообщение, говорящее о том, что итератор не поддерживает концепт `random_access_iterator`.



Стандартная библиотека шаблонов (Standard Template Library)



Суть обобщенного программирования

Я хочу начать это краткое историческое отступление с цитаты из бесценной книги «*От математики к обобщенному программированию*»¹, написанной Александром Степановым (создателем стандартной библиотеки шаблонов) и Даниэлем Розе (исследователем по извлечению информации): «Суть обобщенного программирования лежит в идее концептов. Концепт – это способ описания семейства связанных типов объектов». Эти связанные типы могут быть такими типами, как `bool`, `char` или `int`. Концепт олицетворяет множество требований к связанным типам, таких как поддерживаемые операции, семантика и сложность по времени и памяти.

Стандартная библиотека шаблонов (STL) как обобщенная библиотека основана на концептах. С одной стороны, STL состоит из трех компонентов. Ими являются контейнеры, алгоритмы, работающие над контейнерами, и итераторы, соединяющие первые два.

¹ <https://www.fm2gp.com>.

Каждый контейнер предоставляет итераторы, которые поддерживают его структуру, и алгоритмы, работающие над этими итераторами. Контейнер, последовательный или ассоциативный, моделирует полуоткрытый диапазон. Доступ к элементам контейнера предоставляется через итераторы, итерирование через них и сравнение их на равенство. Абстракция STL основана на концептах, таких как полуоткрытый диапазон и итератор, и позволяет прозрачно использовать контейнеры и алгоритмы STL.

Если посмотреть более общо, то в чем заключаются преимущества концептов?

4.1.2 Преимущества концептов

- Требования к параметрам шаблона являются частью интерфейса.
- Перегрузка функций и специализация шаблонов классов также может быть основана на концептах.
- Концепты могут быть использованы для шаблонных функций, шаблонных классов и универсальных (generic) функций классов или шаблонных классов.
- Вы получаете более понятные сообщения об ошибках, поскольку компилятор сравнивает требования к шаблонным параметрам с заданными аргументами шаблона.
- Вы можете использовать предопределенные концепты и создавать свои собственные.
- Единое использование `auto` и концептов. Вместо `auto` вы можете использовать концепты.
- Если объявление функции использует концепты, то она автоматически становится шаблонной функцией. Таким образом, написание шаблонной функции становится таким же простым, как и написание обычной функции.

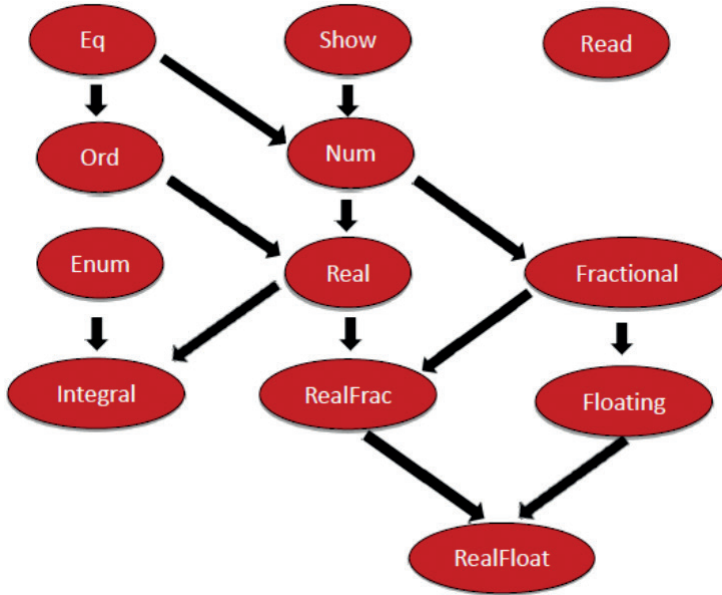
4.1.3 Длинная, длинная история

Первый раз, когда я услышал про концепты, шел примерно 2005–2006 год. Они напомнили мне типовые классы в Haskell. Типовые классы в Haskell – это интерфейсы для схожих типов. Ниже приводится фрагмент иерархии типовых классов в Haskell¹.

Но концепты в C++ – это кое-что другое. Ниже приводятся их некоторые отличия:

- В Haskell любой тип должен быть экземпляром типового класса. В C++20 тип просто должен удовлетворять требованиям концепта.
- Концепты могут применяться к аргументам шаблонов, которые не являются типами. Например, числа, такие как 5, являются примерами подобных аргументов. Когда вам нужен `std::array` из 5 `int`, то вы используете аргумент, не являющийся типом: `std::array<int, 5> myArray`.
- Концепты не добавляют дополнительно задержки ко времени выполнения программы.

¹ [https://en.wikipedia.org/wiki/Haskell_\(programming_language\)](https://en.wikipedia.org/wiki/Haskell_(programming_language)).



Иерархия типовых классов в Haskell

Изначально концепты должны были быть одной из ключевых возможностей C++11, но они были убраны во время встречи по стандартизации во Франкфурте. Высказывание Бьярна Страуструпа говорит само за себя: «Дизайн концептов в C++0x породил монстра сложности»¹. Несколькими годами позже следующая попытка также не была успешна: концепты были убраны из стандарта C++17. И они стали частью языка только в стандарте C++20.

4.1.4 Использование концептов

На самом деле есть четыре способа использования концептов.

4.1.4.1 Четыре пути использования концептов

Я использую предопределенный концепт `std::integral` в программе `conceptsIntegralVariations.cpp` всеми четырьмя способами.

¹ <https://isocpp.org/blog/2013/02/concepts-lite-constraining-templates-with-predicates-andrew-sutton-bjarne-s>.

Четыре варианта использования концепта `std::integral`

```
1  // conceptsIntegralVariations.cpp
2
3  #include <concepts>
4  #include <iostream>
5
6  template<typename T>
7  requires std::integral<T>
8  auto gcd(T a, T b) {
9      if( b == 0 ) return a;
10     else return gcd(b, a % b);
11 }
12
13 template<typename T>
14 auto gcd1(T a, T b) requires std::integral<T> {
15     if( b == 0 ) return a;
16     else return gcd1(b, a % b);
17 }
18
19 template<std::integral T>
20 auto gcd2(T a, T b) {
21     if( b == 0 ) return a;
22     else return gcd2(b, a % b);
23 }
24
25 auto gcd3(std::integral auto a, std::integral auto b) {
26     if( b == 0 ) return a;
27     else return gcd3(b, a % b);
28 }
29
30 int main(){
31
32     std::cout << '\n';
33
34     std::cout << "gcd(100, 10)= " << gcd(100, 10) << '\n';
35     std::cout << "gcd1(100, 10)= " << gcd1(100, 10) << '\n';
36     std::cout << "gcd2(100, 10)= " << gcd2(100, 10) << '\n';
37     std::cout << "gcd3(100, 10)= " << gcd3(100, 10) << '\n';
38 }
```

```

39     std::cout << '\n';
40
41 }

```

Благодаря заголовочному файлу `<concepts>` в строке 3 я могу использовать концепт `std::integral`. Этот концепт удовлетворен, если тип `T` является целочисленным¹. Используемое имя функции `gcd` обозначает алгоритм наибольшего целочисленного делителя, основанного на алгоритме Евклида².

Пример выше демонстрирует четыре способа использования концептов:

- директива `requires` (строка 6);
- директива `requires`, идущая за объявлением функции (строка 13);
- параметр шаблона с ограничением (строка 19);
- сокращенный шаблон функции (строка 25).

Для простоты результат каждой функции обозначен как `auto`. Есть определенная семантическая разница между шаблонными функциями `gcd`, `gcd1`, `gcd2` и `gcd3`. В случае `gcd`, `gcd1` или `gcd2` оба аргумента `a` и `b` должны быть одного типа. Но это не обязательно для функции `gcd3`. Параметры `a` и `b` могут быть разных типов, но оба должны реализовывать концепт `std::integral`.

```

gcd(100, 10) = 10
gcd1(100, 10) = 10
gcd2(100, 10) = 10
gcd3(100, 10) = 10

```

Использование концепта `std::integral`

Функции `gcd` и `gcd1` используют директивы `requires`. Директивы `requires` обладают большими возможностями, чем вы могли подумать. Давайте более подробно познакомимся с ними.

4.1.4.2 Директива `requires`

В предыдущей программе `conceptsIntegralVariations.cpp` показано, как можно использовать концепт для определения обычной или шаблонной функции. На самом деле есть гораздо больше различных случаев. Для полноты нужно добавить, что при помощи концептов вы можете задавать также возвращаемое функцией значение.

Ключевое слово `requires` вводит директиву `requires`, которая задает условия на аргумент шаблона (`gcd`) или объявления функции (`gcd1`). За `requires` должен идти предикат времени компиляции, такой как именованный концепт (`gcd`), конъюнкция или дизъюнкция именованных концептов и `requires-выражение`.

Предикат времени компиляции также может быть выражением.

¹ https://en.cppreference.com/w/cpp/types/is_integral.

² <https://en.wikipedia.org/wiki/Euclid>.

Использование предиката времени компиляции в директиве `requires`

```

1  // requiresClause.cpp
2
3  #include <iostream>
4
5  template <unsigned int i>
6  requires (i <= 20)
7  int sum(int j) {
8      return i + j;
9  }
10
11
12  int main() {
13
14      std::cout << '\n';
15
16      std::cout << "sum<20>(2000): " << sum<20>(2000) << '\n',
17      // std::cout << "sum<23>(2000): " << sum<23>(2000) << '\n', // ERROR
18
19      std::cout << '\n';
20
21  }
```

Предикат времени компиляции, использованный в строке 6, иллюстрирует интересную возможность: требование накладывается не на тип, а на само значение `i`.

```
sum<20>(2000) : 2020
```

Предикат времени компиляции в директиве `requires`

При обработке строки 17 clang выдает следующую ошибку:

```

<source>:17:39: error: no matching function for call to 'sum'
    std::cout << "sum<23>(2000): " << sum<23>(2000) << '\n', // ERROR
                                   ^~~~~~
<source>:7:5: note: candidate template ignored: constraints not satisfied [with i = 23]
int sum(int j) {
  ^
<source>:6:11: note: because '23U <= 20' (23 <= 20) evaluated to false
requires (i <= 20)
      ^
```

Невыполнение предиката времени компиляции



Избегайте предикатов времени компиляции в директивах `requires`

Когда вы накладываете ограничения на параметр шаблона или шаблонной функции, вам лучше использовать именованные концепты или их комбинацию. Концепты должны быть семантическими категориями, но не синтаксическими ограничениями типа `i <= 20`. Задание имен для концептов облегчает их повторное использование.

4.1.4.3 Концепты как возвращаемое значение функции

Ниже приводятся определения шаблонной функции `gcd` и функции `gcd1`, использующих концепты для задания возвращаемого типа.

Использование концепта для задания возвращаемого типа

```
template<typename T>
requires std::integral<T>
std::integral auto gcd(T a, T b) {
    if( b == 0 ) return a;
    else return gcd(b, a % b);
}

std::integral auto gcd1(std::integral auto a, std::integral auto b) {
    if( b == 0 ) return a;
    else return gcd1(b, a % b);
}
```

4.1.4.4 Примеры использования концептов

Самое важное – это то, что концепты являются предикатами времени компиляции. Предикат времени компиляции – это функция, которая выполняется во время компиляции и возвращает логическое значение. Прежде чем я начну рассматривать различные примеры использования концептов, я хочу лишить их таинственности и показать, что это просто функции, возвращающие логические значения во время компиляции.

4.1.4.4.1 Предикаты времени компиляции

Концепт может быть использован в управляющей структуре.

Концепты как предикаты времени компиляции

```
1 // compileTimePredicate.cpp
2
3 #include <compare>
4 #include <iostream>
5 #include <string>
6 #include <vector>
7
```

```
8  struct Test{};
9
10 int main() {
11
12     std::cout << '\n';
13
14     std::cout << std::boolalpha;
15
16     std::cout << "std::three_way_comparable<int>: "
17                 << std::three_way_comparable<int> << "\n";
18
19     std::cout << "std::three_way_comparable<double>: ";
20     if (std::three_way_comparable<double>) std::cout << "True";
21     else std::cout << "False";
22
23     std::cout << "\n\n";
24
25     static_assert(std::three_way_comparable<std::string>);
26
27     std::cout << "std::three_way_comparable<Test>: ";
28     if constexpr(std::three_way_comparable<Test>) std::cout << "True";
29     else std::cout << "False";
30
31     std::cout << '\n';
32
33     std::cout << "std::three_way_comparable<std::vector<int>>: ";
34     if constexpr(std::three_way_comparable<std::vector<int>>)
35                                     std::cout << "True";
36     else std::cout << "False";
37
38     std::cout << '\n';
39 }
```

В приведенной выше программе я использую концепт `std::three_way_comparable<T>`, который во время компиляции проверяет на тип `T` шесть операторов сравнения. Поскольку это предикат времени компиляции, то он может быть использован как во время выполнения (строки 16 и 20), так и во время компиляции. Конструкции `static_assert` (строка 25) и `constexpr if`¹ (строки 28 и 34) вычисляются во время компиляции.

¹ <https://en.cppreference.com/w/cpp/language/if>.


```
std::three_way_comparable<int>: true
std::three_way_comparable<double>: True

std::three_way_comparable<Test>: False
std::three_way_comparable<std::vector<int>>: True
```

Концепты как предикаты времени компиляции

После того как было показано, что концепты являются предикатами времени компиляции, давайте продолжим рассмотрение различных случаев использования концептов. Примеры использования концептов не будут слишком сложными и в основном будут опираться на predefined концепты, которые я более детально опишу в соответствующем разделе.

4.1.4.4.2 Шаблоны классов

Шаблонный класс `MyVector` требует, чтобы его параметр шаблона `T` был `regular`, т. е. вел себя примерно как `int`. Формальное определение `regular` будет приведено позже.

Использование концепта при определении класса

```
1  // conceptsClassTemplate.cpp
2
3  #include <concepts>
4  #include <iostream>
5
6  template <std::regular T>
7  class MyVector{};
8
9  int main() {
10
11     MyVector<int> myVec1;
12     MyVector<int&> myVec2; // ERROR because a reference
13                           // is not regular
14 }
```

Строка 12 вызывает ошибку времени компиляции, поскольку ссылка не является `regular`. Ниже приводится основная часть сообщения об ошибке GCC.

```
<source>:13:18: error: template constraint failure for 'template<class T> requires regular<T> class MyVector'
13 |     MyVector<int&> myVec2;
```

Ссылка не `regular`

4.1.4.4.3 Функции-члены, не являющиеся шаблонными

В этом примере я рассмотрю функцию-член `push_back` класса `MyVector`. Этот метод требует, чтобы аргумент можно было скопировать.

Использование концепта в функции-члене

```
1  // conceptMemberFunction.cpp
2
3  #include <concepts>
4  #include <iostream>
5
6  struct NotCopyable {
7      NotCopyable() = default;
8      NotCopyable(const NotCopyable&) = delete;
9  };
10
11 template <typename T>
12 struct MyVector{
13     void push_back(const T&) requires std::copyable<T> {}
14 };
15
16 int main() {
17
18     MyVector<int> myVec1;
19     myVec1.push_back(2020);
20
21     MyVector<NotCopyable> myVec2;
22     myVec2.push_back(NotCopyable()); // ERROR because
23                                     // not copyable
24 }
```

Компиляция этого примера приводит к ошибке в строке 22. Экземпляры `NotCopyable` нельзя копировать, поскольку в них удален конструктор копирования.

4.1.4.4.4 Шаблоны с переменным количеством аргументов (variadic templates)

Вы также можете использовать концепты в шаблонах с переменным количеством аргументов.

Применение концептов в шаблонах с переменным количеством аргументов

```

1  // allAnyNone.cpp
2
3  #include <concepts>
4  #include <iostream>
5
6  template<std::integral... Args>
7  bool all(Args... args) { return (... && args); }
8
9  template<std::integral... Args>
10 bool any(Args... args) { return (... || args); }
11
12 template<std::integral... Args>
13 bool none(Args... args) { return not(... || args); }
14
15 int main(){
16
17     std::cout << std::boolalpha << '\n';
18
19     std::cout << "all(5, true, false): " << all(5, true, false) << '\n';
20
21     std::cout << "any(5, true, false): " << any(5, true, false) << '\n';
22
23     std::cout << "none(5, true, false): " << none(5, true, false) << '\n';
24
25 }
```

Определение приведенных выше шаблонов основано на свертке параметров шаблона (fold expression). C++11 поддерживает шаблоны с произвольным переменным количеством аргументов. Произвольное количество шаблонных параметров содержится в так называемом *parameter pack* – совокупности нулевого или произвольного количества типов. Кроме того, в C++ вы можете применить операцию редукции к этой совокупности при помощи произвольного бинарного оператора. Подобная редукция называется свертка параметров шаблона¹. В этом примере логическое И && (строка 7), логическое ИЛИ || (строка 10) и отрицание логического ИЛИ (строка 13) применяются как бинарные операторы. Кроме того, *all*, *any* и *none* требуют от своих параметров поддержки концепта *std::integral*.

```

all(5, true, false): false
any(5, true, false): true
none(5, true, false): false
```

Применение концептов к свертке параметров шаблона

¹ <https://www.modernescpp.com/index.php/fold-expressions>.

4.1.4.4.5 Перегрузка

`std::advance`¹ – это алгоритм в стандартной библиотеке шаблонов. Он сдвигает заданный итератор `iter` на `n` элементов. Исходя из возможностей, итераторы могут применять различные стратегии. Например, `std::forward_list` поддерживает итератор, который может передвигаться только в одном направлении, в то время как `std::list` поддерживает итератор, который может передвигаться в обоих направлениях, а `std::vector` поддерживает итератор с произвольным доступом. Соответственно, для итератора, предоставленного `std::forward_list` или `std::list`, вызов `std::advance(iter, n)` должен выполнить `n` операций инкремента (см. структуру `std::list`). Эта сложность по времени отличается от сложности по времени для `std::random_access_iterator`, предоставляемого `std::vector`. В последнем случае мы просто прибавляем к итератору число `n`. Таким образом, линейная сложность по времени $O(n)$ превращается в константную сложность $O(1)$. Мы можем использовать концепты для того, чтобы различать различные типы итераторов. Программа `conceptsOverloadingFunctionTemplates.cpp` призвана дать вам понимание этого.

Перегрузка шаблонных функций с использованием концептов

```
1 // conceptsOverloadingFunctionTemplates.cpp
2
3 #include <concepts>
4 #include <iostream>
5 #include <forward_list>
6 #include <list>
7 #include <vector>
8
9 template<std::forward_iterator I>
10 void advance(I& iter, int n){
11     std::cout << "forward_iterator" << '\n';
12 }
13
14 template<std::bidirectional_iterator I>
15 void advance(I& iter, int n){
16     std::cout << "bidirectional_iterator" << '\n';
17 }
18
19 template<std::random_access_iterator I>
20 void advance(I& iter, int n){
21     std::cout << "random_access_iterator" << '\n';
22 }
23
```

¹ <https://en.cppreference.com/w/cpp/iterator/advance>.

```

24  int main() {
25
26      std::cout << '\n';
27
28      std::forward_list forwList{1, 2, 3};
29      std::forward_list<int>::iterator itFor = forwList.begin();
30      advance(itFor, 2);
31
32      std::list li{1, 2, 3};
33      std::list<int>::iterator itBi = li.begin();
34      advance(itBi, 2);
35
36      std::vector vec{1, 2, 3};
37      std::vector<int>::iterator itRa = vec.begin();
38      advance(itRa, 2);
39
40      std::cout << '\n';
41  }

```

Три варианта функции `advance` перегружены, исходя из концептов `std::forward_iterator` (строка 9), `std::bidirectional_iterator` (строка 14) и `std::random_access_iterator` (строка 19). Компилятор сам выберет наиболее подходящий вариант. Это значит, что для `std::forward_list` (строка 28) будет выбран вариант, удовлетворяющий концепту `std::forward_iterator`, для `std::list` (строка 32) будет выбран перегруженный вариант, исходя из концепта `std::bidirectional_iterator`, и для `std::vector` (строка 36) будет выбран перегруженный вариант, основанный на концепте `std::random_access_iterator`.

```

forward_iterator
bidirectional_iterator
random_access_iterator

```

Перегрузка функций при помощи концептов

Это происходит, поскольку `std::random_access_iterator` является `std::bidirectional_iterator`, а `std::bidirectional_iterator`, в свою очередь, является `std::forward_iterator`.

4.1.4.4.6 Специализация шаблонов

При помощи концептов также можно специализировать шаблоны.

Специализация шаблонов при помощи концептов

```
1  // conceptsSpecialization.cpp
2
3  #include <concepts>
4  #include <iostream>
5
6  template <typename T>
7  struct Vector {
8      Vector() {
9          std::cout << "Vector<T>" << '\n';
10     }
11 };
12
13 template <std::regular Reg>
14 struct Vector<Reg> {
15     Vector() {
16         std::cout << "Vector<std::regular>" << '\n';
17     }
18 };
19
20 int main() {
21
22     std::cout << '\n';
23
24     Vector<int> myVec1;
25     Vector<int&> myVec2;
26
27     std::cout << '\n';
28
29 }
```

При инстанцииции шаблона класса компилятор выбирает наиболее подходящий вариант. Это значит, что для `Vector<int> myVec` (строка 24) будет выбрана частичная специализация, основанная на `std::regular` (строка 13). А вот для ссылочного `Vector<int&> myVec2` (строка 25) тип не является регулярным (`regular`), и поэтому будет выбрана базовая версия (строка 6).

```
Vector<std::regular>
Vector<T>
```

4.1.4.4.7 Использование нескольких концептов

До сих пор применение концептов было крайне простым и одновременно использовалось не более одного концепта.

Использование более чем одного концепта

```
template<typename Iter, typename Val>
    requires std::input_iterator<Iter>
           && std::equality_comparable<Value_type<Iter>, Val>
Iter find(It b, It e, Val v)
```

Функция `find` требует, чтобы итератор `Iter` и его сравнение с заданным значением `Val` было выполнено при следующих условиях:

- `Iter` должен быть входным итератором;
- значение, соответствующее итератору `Iter`, должно быть сравнимо на равенство с `Val`.

Это же требование к итератору может быть выражено при помощи ограничивающего шаблонного параметра.

Использование более чем одного концепта

```
template<std::input_iterator Iter, typename Val>
    requires std::equality_comparable<Value_type<Iter>, Val>
Iter find(It b, It e, Val v)
```

4.1.5 Ограниченные или неограниченные заполнители

Сначала давайте поговорим об асимметрии в C++14.

4.1.5.1 Большая асимметрия в C++14

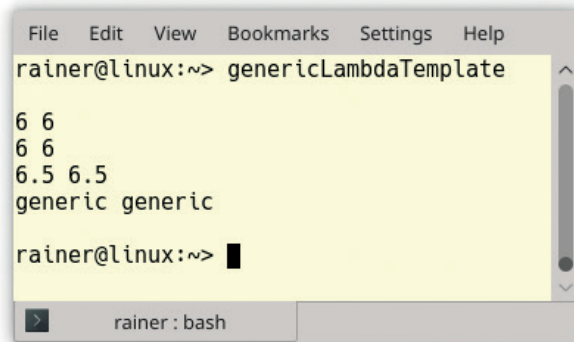
При ведении занятий я часто сталкиваюсь со следующим обсуждением. В стандарте C++14 у нас есть обобщенные лямбда-функции (generic lambda). Обобщенные лямбды используют `auto` вместо указания конкретного типа.

Сравнение обобщенных лямбд и шаблонных функций

```
1 // genericLambdaTemplate.cpp
2
3 #include <iostream>
4 #include <string>
5
6 auto addLambda = [](auto fir, auto sec){ return fir + sec; };
7
8 template <typename T, typename T2>
9 auto addTemplate(T fir, T2 sec){ return fir + sec; }
10
```

```
11 int main(){
12
13     std::cout << std::boolalpha << '\n';
14
15     std::cout << addLambda(1, 5) << " " << addTemplate(1, 5) << '\n';
16     std::cout << addLambda(true, 5) << " " << addTemplate(true, 5) << '\n';
17     std::cout << addLambda(1, 5.5) << " " << addTemplate(1, 5.5) << '\n';
18
19     const std::string fir{"ge"};
20     const std::string sec{"neric"};
21     std::cout << addLambda(fir, sec) << " " << addTemplate(fir, sec) << '\n';
22
23     std::cout << '\n';
24
25 }
```

Обобщенная лямбда (строка 6) и шаблонная функция (строка 8) дают одинаковые результаты.



```
File Edit View Bookmarks Settings Help
rainer@linux:~> genericLambdaTemplate
6 6
6 6
6.5 6.5
generic generic
rainer@linux:~> 
```

Использование обобщенной лямбды и шаблонной функции

Обобщенная лямбда дает новый способ задания шаблонных функций. На своих занятиях я часто спрашиваю: можем ли мы использовать `auto` для получения шаблонов функций? Для C++14 ответ – нет, но для C++20 ответ – да. В C++20 вы можете использовать заполнитель без ограничений (unconstrained placeholders)(`auto`) или заполнитель с ограничением (constrained placeholder) (концепт) в описании функции для автоматического получения шаблонной функции. Правило применения очень простое. В каждом месте, где вы можете использовать `auto`, вы можете использовать концепт. Более подробно я рассмотрю это в следующем разделе, посвященном сокращенным шаблонным функциям (abbreviated function templates).

4.1.5.2 Заполнители

Использование заполнителя с ограничением (constrained placeholder) вместо заполнителя без ограничений (Unconstrained Placeholder)

```

1  // placeholders.cpp
2
3  #include <concepts>
4  #include <iostream>
5  #include <vector>
6
7  std::integral auto getIntegral(int val){
8      return val;
9  }
10
11 int main(){
12
13     std::cout << std::boolalpha << '\n';
14
15     std::vector<int> vec{1, 2, 3, 4, 5};
16     for (std::integral auto i: vec) std::cout << i << " ";
17     std::cout << '\n';
18
19     std::integral auto b = true;
20     std::cout << b << '\n';
21
22     std::integral auto integ = getIntegral(10);
23     std::cout << integ << '\n';
24
25     auto integ1 = getIntegral(10);
26     std::cout << integ1 << '\n';
27
28     std::cout << '\n';
29
30 }
```

Концепт `std::integral` может использоваться как возвращаемый тип (строка 7) в цикле по контейнеру (строка 16) или как тип переменной `b` (строка 19) либо как тип переменной `integ` (строка 22). Для того чтобы увидеть симметрию между `auto` и концептами, строка 25 использует `auto` вместо `std::integral auto`, как в строке 22. Поэтому `integ1` может принять значение любого типа.

```
1 2 3 4 5
true
10
10
```

Заполнитель с ограничением вместо заполнителя без ограничения в действии

4.1.6 Сокращенные шаблонные функции

В C++20 вы можете использовать заполнитель без ограничений (`auto`) или заполнитель с ограничением (концепт) в описании функции, и тогда описание этой функции автоматически становится шаблоном.

Сокращенные шаблонные функции

```
1  // abbreviatedFunctionTemplates.cpp
2
3  #include <concepts>
4  #include <iostream>
5
6  template<typename T>
7  requires std::integral<T>
8  T gcd(T a, T b) {
9      if( b == 0 ) return a;
10     else return gcd(b, a % b);
11 }
12
13 template<typename T>
14 T gcd1(T a, T b) requires std::integral<T> {
15     if( b == 0 ) return a;
16     else return gcd1(b, a % b);
17 }
18
19 template<std::integral T>
20 T gcd2(T a, T b) {
21     if( b == 0 ) return a;
22     else return gcd2(b, a % b);
23 }
24
25 std::integral auto gcd3(std::integral auto a, std::integral auto b) {
26     if( b == 0 ) return a;
27     else return gcd3(b, a % b);
28 }
29
```

```

30 auto gcd4(auto a, auto b){
31     if( b == 0 ) return a;
32     return gcd4(b, a % b);
33 }
34
35 int main() {
36
37     std::cout << '\n';
38
39     std::cout << "gcd(100, 10)= " << gcd(100, 10) << '\n';
40     std::cout << "gcd1(100, 10)= " << gcd1(100, 10) << '\n';
41     std::cout << "gcd2(100, 10)= " << gcd2(100, 10) << '\n';
42     std::cout << "gcd3(100, 10)= " << gcd3(100, 10) << '\n';
43     std::cout << "gcd4(100, 10)= " << gcd4(100, 10) << '\n';
44
45     std::cout << '\n';
46
47 }

```

Определения шаблонных функций gcd (строка 6), gcd1 (строка 13) и gcd2 (строка 19) полностью совпадают с теми, которые я уже показывал в разделе «Четыре способа использования концептов». Функция gcd использует директиву requires, gcd1 использует директиву requires, идущую за объявлением функции, и gcd2 – параметр шаблона с ограничением. Теперь посмотрим, что появилось нового. Шаблон gcd3 применяет концепт std::integral как параметр типа и становится шаблонной функцией с ограниченным параметром типа. В отличие от этого, gcd4 эквивалентна шаблонной функции без ограничений на его типы. Используемый в gcd3 и gcd4 синтаксис для создания шаблонных функций называется сокращенной шаблонной функцией.

```

gcd(100, 10)= 10
gcd1(100, 10)= 10
gcd2(100, 10)= 10
gcd3(100, 10)= 10
gcd4(100, 10)= 10

```

Ограничения

Позвольте мне подчеркнуть эту симметрию демонстрацией еще одного примера.

Используя auto как параметр типа, функция add становится шаблонной функцией и полностью эквивалентна одноименной шаблонной функции add.

Эквивалентные функция и шаблонная функция add

```
template<typename T, typename T2>
auto add(T fir, T2 sec) {
    return fir + sec;
}
```

```
auto add(auto fir, auto sec) {
    return fir + sec;
}
```

Соответственно, в связи с использованием концепта `std::integral` функция `sub` эквивалентна шаблонной функции `sub`.

Эквивалентные функция и шаблонная функция

```
template<std::integral T, std::integral T2>
std::integral auto sub(T fir, T2 sec) {
    return fir - sec;
}
```

```
std::integral auto sub(std::integral auto fir, std::integral auto sec) {
    return fir - sec;
}
```

И функция, и шаблонная функция могут иметь произвольные типы. Это значит, что оба типа могут быть разными, но оба обязаны быть целочисленными. Например, вызовы `sub(100,10)` и `sub(100, true)` будут корректными.

Есть еще одна интересная возможность, пока отсутствующая в синтаксисе сокращенных шаблонных функций: вы можете перегрузить функции, используя `auto` или концепты.

4.1.6.1 Перегрузка

Следующие функции `overload` являются перегруженными при помощи `auto`, концепта `std::integral` и типа `long`.

Сокращенные шаблонные функции и перегрузка

```
1 // conceptsOverloading.cpp
2
3 #include <concepts>
4 #include <iostream>
5
6 void overload(auto t){
7     std::cout << "auto : " << t << '\n';
8 }
```

```

9
10 void overload(std::integral auto t){
11     std::cout << "Integral : " << t << '\n';
12 }
13
14 void overload(long t){
15     std::cout << "long : " << t << '\n';
16 }
17
18 int main(){
19
20     std::cout << '\n';
21
22     overload(3.14);
23     overload(2010);
24     overload(2020L);
25
26     std::cout << '\n';
27
28 }

```

Компилятор выбирает перегруженный вариант с `auto` (строка 6) для `double`, перегруженный вариант с `std::integral` (строка 10) для `int` и перегруженный для `long` вариант (строка 14) для типа `long`.

```

auto : 3.14
Integral : 2010
long : 2020

```

Сокращенные шаблонные функции и перегрузка



Чего у нас не было ранее: введение шаблонов

Возможно, вам не хватает одного элемента в этой главе по концептам – введения шаблонов (template introduction). Введение шаблонов было частью технической спецификации (TS) по концептам, TS ISO/IEC TS 19217-2015¹ и экспериментальной реализацией концептов. GCC6² полностью реализует техническую спецификацию по концептам. Не считая синтаксических отличий от концептов из C++20, техническая спецификация по концептам поддерживает сокращенный способ определения шаблонов.

¹ <https://www.iso.org/standard/64031.html>.

² https://en.wikipedia.org/wiki/GNU_Compiler_Collection.

В следующем примере мы будем считать, что `Integral` – это концепт.

Введение шаблонов в техническую спецификацию по концептам

```
Integral{T}
Integral gcd(T a, T b){
    if( b == 0 ){ return a; }
    else{
        return gcd(b, a % b);
    }
}
```

```
Integral{T}
class ConstrainedClass{};
```

Этот небольшой пример кода использует введение шаблонов двумя способами: для определения шаблонной функции с параметром с ограничением и для определения шаблонного класса с параметром с ограничением. Введение шаблонов обладает одним ограничением. Вы можете использовать его только с параметром шаблона с ограничением (концепт), но не с традиционным параметром без ограничений (`auto`). Такая несимметрия может быть легко преодолена при помощи концепта, всегда возвращающего `true`.

Концепт `Generic`, который всегда выполняется

```
template<typename T>
concept bool Generic(){
    return true;
}
```

Я использовал в этом примере синтаксис из технической спецификации для определения концепта `Generic`. Синтаксис в C++20 немного более краткий. Вы можете прочитать о деталях синтаксиса C++20 в разделе «Определение концептов».

4.1.7 Предопределенные концепты

Золотое правило «не изобретай колесо» также применимо и к концептам. В «Руководящих принципах C++»¹ на эту тему написано: «Т.11: Когда возможно, используйте стандартные концепты». Поэтому я хочу дать обзор наиболее важных предопределенных концептов. Я намеренно опускаю некоторые специальные или дополнительные концепты.

Все предопределенные концепты описаны в последней версии C++20 N4860², и найти их может быть очень сложной задачей. Большинство из концептов приведены в главе 18 (библиотека концептов) и главе 24 (библиотека диапазонов). Кроме того, некоторые из концептов приведены в главе 17 (библиотека

¹ <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>.

² <https://isocpp.org/files/papers/N4860.pdf>.

поддержки языка), главе 20 (библиотека общих утилит), главе 23 (библиотека итераторов) и главе 26 (библиотека численных методов). В черновике стандарта C++20 N4860 содержится список всех библиотечных концептов с описанием того, как они реализованы.

4.1.7.1 Библиотека поддержки языка

В этом разделе рассматривается интересный концепт `three_way_comparable`. Он используется для поддержки оператора трехстороннего сравнения (`three-way comparison operator`) и определен в заголовочном файле `<compare>`.

Рассмотрим более формальное определение. Пусть `a` и `b` – это значения типа `T`. Эти значения `three_way_comparable`, если:

- `(a <=> b == 0) == bool(a == b)` равно `true`
- `(a <=> b != 0) == bool(a != b)` равно `true`
- `((a <=> b) <=> 0)` и `(0 <=> (b <=> a))` равны
- `(a <=> b < 0) == bool(a < b)` равно `true`
- `(a <=> b > 0) == bool(a > b)` равно `true`
- `(a <=> b <= 0) == bool(a <= b)` равно `true`
- `(a <=> b >= 0) == bool(a >= b)` равно `true`

4.1.7.2 Библиотека концептов

Наиболее часто используемые концепты можно найти в библиотеке концептов. Они определены в заголовочном файле `<concepts>`.

4.1.7.2.1 Лексически понятные концепты

В этом разделе содержится 15 концептов, которые понятны сами по себе. Эти концепты выражают отношения между типами, задают классификацию типов и базовые свойства типов. Их реализация часто основана на соответствующих функциях из библиотеки признаков типа (`type-traits library`)¹. Там, где это важно, я даю дополнительные пояснения.

- `same_as`
- `derived_from`
- `convertible_to`
- `common_reference_with`: `common_reference_with<T, U>` – тип данных должен быть правильно построенным (`well-formed`), а для типов `T` и `U` должна быть обеспечена возможность конвертации в ссылочный тип `S`, где `S` совпадает с `common_reference_t<T, U>`
- `common_with` аналогичен `common_reference_with`, но при этом общий тип `S` совпадает с `common_type_t<T, U>` и не может быть ссылочным типом
- `assignable_from`
- `swappable`

4.1.7.2.2 Арифметические концепты

- `integral`
- `signed_integral`
- `unsigned_integral`
- `floating_point`

¹ https://en.cppreference.com/w/cpp/header/type_traits.

В стандарте содержится прямое определение арифметических концептов:

```
template<class T>
```

```
concept integral = is_integral_v<T>;
```

```
template<class T>
```

```
concept signed_integral = integral<T> && is_signed_v<T>;
```

```
template<class T>
```

```
concept unsigned_integral = integral<T> && !signed_integral<T>;
```

```
template<class T>
```

```
concept floating_point = is_floating_point_v<T>;
```

4.1.7.2.3 Концепты времени жизни

- destructible
- constructible_from
- default_constructible
- move_constructible
- copy_constructible

4.1.7.2.4 Концепты сравнения

- equality_comparable
- totally_ordered

Возможно, вы помните с занятий по математике, что T для значений a и b типа T является `totally_ordered` (полностью упорядоченным) тогда и только тогда, когда:

- только одно из следующих `bool(a < b)`, `bool(a > b)` и `bool(a == b)` равно `true`;
- если `bool(a < b)` и `bool(b < c)`, то `bool(a < c)`;
- `bool(a > b) == bool(b < a)`;
- `bool(a <= b) == !bool(b < a)`;
- `bool(a >= b) == !bool(a < b)`.

4.1.7.2.5 Концепты объектов

- movable
- copyable
- semiregular
- regular

Ниже приводятся краткие определения всех этих четырех концептов:

```
template<class T>
```

```
concept movable = is_object_v<T> && move_constructible<T> &&  
                  assignable_from<T&, T> && swappable<T>;
```



```

template<class T>
concept copyable = copy_constructible<T> && movable<T> &&
                    assignable_from<T&, T&> &&
                    assignable_from<T&, const T&> && assignable_from<T&, const T>;

template<class T>
concept semiregular = copyable<T> && default_initializable<T>;

template<class T>
concept regular = semiregular<T> && equality_comparable<T>;

```

Концепт `movable` требует для `T`, чтобы выполнялось `is_object_v<T>`. Из определения признака типа `is_object<T>` следует, что тип `T` может быть скалярным значением, массивом, объединением (`union`) или классом.

Я реализую концепты `semiregular` и `regular` в разделе «определение концептов». Фактически `semiregular` означает, что тип `T` ведет себя как `int`. А `regular` означает, что тип ведет себя как `int` и может быть сравнен на равенство при помощи оператора «`==`».

4.1.7.2.6 Вызываемые концепты

- `invocable`
- `regular_invocable`: тип, похожий на `invocable`, но не изменяющий аргументы функции и поддерживающий защиту эквивалентности, которая означает, что всегда будет один и тот же результат при одних и тех же входных данных;
- `predicate`: тип, моделирующий предикат, если он реализует вызываемый тип данных и возвращает значение типа `boolean`.

4.1.7.3 Библиотека общих утилит

В этой части стандарта содержатся описания только специальных концептов, связанных с памятью; поэтому я не буду их здесь упоминать.

4.1.7.4 Библиотека итераторов

В библиотеке итераторов содержится много важных концептов. Они определены в заголовочном файле `<iterator>`. Ниже приводятся типы этих итераторов:

- `input_iterator`
- `output_iterator`
- `forward_iterator`
- `bidirectional_iterator`
- `random_access_iterator`
- `contiguous_iterator`

Эти шесть типов итераторов соответствуют шести концептам. В таблице ниже приводятся свойства и соответствующие контейнеры для трех наиболее важных типов.

Тип итераторов	Свойства	Контейнеры
<code>std::forward_iterator</code>	<code>++It, It++, *It</code> <code>It == It2, It != It2</code>	<code>std::unordered_set</code> <code>std::unordered_map</code> <code>std::unordered_multiset</code> <code>std::unordered_multimap</code> <code>std::forward_list</code>
<code>std::bidirectional_iterator</code>	<code>--It, It--</code>	<code>std::set</code> <code>std::map</code> <code>std::multiset</code> <code>std::multimap</code> <code>std::list</code>
<code>std::random_access_iterator</code>	<code>It[i]</code> <code>It += n, It -= n</code> <code>It + n, It - n</code> <code>n + It</code> <code>It - It2</code> <code>It < It2, It <= It2</code> <code>It > It2, It >= It2</code>	<code>std::array</code> <code>std::vector</code> <code>std::deque</code> <code>std::string</code>

Справедливы следующие отношения: итератор с произвольным доступом является двунаправленным итератором, и двунаправленный итератор является итератором прямого доступа. Непрерывный итератор – это итератор с произвольным доступом, требующий, чтобы элементы контейнера располагались в памяти последовательно. Это значит, что `std::array`, `std::vector` и `std::string`, но не `std::deque`, поддерживают непрерывные итераторы.

4.1.7.4.1 Концепты, используемые для алгоритмов

- `permutable`: возможно переупорядочение элементов;
- `mergeable`: возможно слияние отсортированных последовательностей в выходную отсортированную последовательность;
- `sortable`: возможна перестановка элементов для превращения последовательности в отсортированную.

4.1.7.5 Библиотека диапазонов

Библиотека диапазонов содержит ряд концептов, важных для работы с диапазонами (`ranges`) и видами (`views`). Они аналогичны концептам в библиотеке итераторов и определены в заголовочном файле `<ranges>`.

4.1.7.5.1 Диапазоны (`range concepts`)

- `range`: диапазон задает группу элементов, по которой можно выполнять итерирование. Он содержит итератор `begin` и маркер конца `end`. Все контейнеры библиотеки STL являются диапазонами.

Также есть дополнительные уточнения для `std::ranges::range`.

- `input_range`: указывает диапазон, чей итератор удовлетворяет `input_iterator` (т. е. можно пройти от начала до конца как минимум один раз).
- `output_range`: задает диапазон, чей итератор удовлетворяет `output_iterator`.
- `forward_range`: задает диапазон, чей итератор удовлетворяет `forward_iterator` (можно пройти от начала до конца более одного раза).

- `bidirectional_range`: задает диапазон, чей итератор удовлетворяет `bidirectional_iterator` (можно перемещаться вперед и назад более одного раза).
- `random_access_range`: задает диапазон, чей итератор удовлетворяет `random_access_iterator` (т. е. за константное время можно перейти к произвольному элементу при помощи оператора `[]`).
- `contiguous_range`: задает диапазон, чей итератор удовлетворяет `contiguous_iterator` (элементы лежат последовательно в одной области памяти).

Каждый контейнер из стандартной библиотеки шаблонов поддерживает соответствующий диапазон. Поддерживаемый диапазон задает свойства своих итераторов.

Свойства и контейнеры для каждого концепта диапазонов

Концепт	Свойства	Контейнеры
<code>std::ranges::input_range</code>	<code>++It, It++, *It</code> <code>It == It2, It != It2</code>	<code>std::unordered_set</code> <code>std::unordered_map</code> <code>std::unordered_multiset</code> <code>std::unordered_multimap</code> <code>std::forward_list</code>
<code>std::ranges::bidirectional_range</code>	<code>--It, It--</code>	<code>std::set</code> <code>std::map</code> <code>std::multiset</code> <code>std::multimap</code> <code>std::list</code>
<code>std::ranges::random_access_range</code>	<code>It[i]</code> <code>It += n, It -= n</code> <code>It + n, It - n</code> <code>n + It</code> <code>It - It2</code> <code>It < It2, It <= It2</code> <code>It > It2, It >= It2</code>	<code>std::deque</code>
<code>std::ranges::contiguous_range</code>	<code>It[i]</code> <code>It += n, It -= n</code> <code>It + n, It - n</code> <code>n + It</code> <code>It - It2</code> <code>It < It2, It <= It2</code> <code>It > It2, It >= It2</code>	<code>std::array</code> <code>std::vector</code> <code>std::string</code>

Контейнер, поддерживающий `std::ranges::contiguous_range`, поддерживает все предыдущие концепты из таблицы, такие как `std::ranges::random_access_range`, `std::ranges::bidirectional_range` и `std::ranges::input_range`. То же самое относится и к другим диапазонам.

4.1.7.5.2 Виды (views)

Под `std::ranges::view` понимается нечто, что вы применяете к диапазону и выполняете над ним некоторую операцию. Вид не владеет данными, а время, затрачиваемое видом на копирование/копию, перемещение или же присваивание, константно. Ниже приводится цитата из реализации `range-v3` Эрика

Ниблера (Eric Niebler), которая была взята за основу построения диапазонов в C++20: «Виды – это адаптация диапазонов, где сама адаптация происходит постепенно по мере итерирования видов».

4.1.7.6 Численная библиотека (numeric library)

Численная библиотека содержит концепт `uniform_random_bit_generator`, который определен в заголовочном файле `<random>`. Он означает, что генератор псевдослучайных чисел `g` типа `G` должен возвращать равномерно распределенные беззнаковые целые числа. Кроме того, генератор `g` типа `G` должен поддерживать функции-члены класса `G::min` и `G::max`.

4.1.8 Определение концептов

Когда нужный вам концепт не входит в число предопределенных концептов в C++20, то вам придется определить свой концепт. В этом разделе я определю несколько концептов, которые будут отличаться от предопределенных использованием синтаксиса `CamelCase`. Соответственно, мой концепт для знаковых целых чисел будет называться `SignedIntegral`, в то время как соответствующий стандартный концепт в C++20 называется `signed_integral`.

Синтаксис для определения концепта очень прост.

Определение концепта

```
template <template-parameter-list>
concept concept-name = constraint-expression;
```

Определение концепта начинается с ключевого слова `template` и содержит список параметров шаблона. Вторая строка определения более интересна – она использует ключевое слово `concept`, за которым идет имя концепта и выражение, задающее соответствующее ограничение.

Часть `constraint-expression` может быть:

- логической комбинацией других концептов или предикатов времени компиляции:
 - ◆ логическая комбинация может быть построена при помощи конъюнкции (`&&`), дизъюнкции (`||`) или отрицания (`!`);
 - ◆ предикаты времени компиляции – это вызываемые сущности, которые возвращают логическое значение во время компиляции;
- выражение `requires` определяет:
 - ◆ простые требования;
 - ◆ требования типов;
 - ◆ составные требования;
 - ◆ вложенные требования.

В следующих двух разделах я покажу различные способы определения концептов.

4.1.8.1 Логическая комбинация других концептов и предикатов времени компиляции

Вы можете объединять концепты и предикаты времени компиляции при помощи конъюнкции (`&&`) и дизъюнкции (`||`). При построении логической ком-

бинации вы можете применять к компонентам отрицание (!). Благодаря наличию большого количества готовых предикатов в библиотеке признаков типа (type-traits library)¹ в вашем распоряжении есть все необходимые инструменты для построения концептов.



Не определяйте концепты рекурсивно и не пытайтесь ограничить их.

Рекурсивное определение концепта недопустимо.

Рекурсивное определение концепта

```
template<typename T>
concept Recursive = Recursive<T*>;
```

В этом случае GCC выдает ошибку, что 'Recursive' was not declared in this scope (Recursive не был определен в этой области видимости).

Когда вы пытаетесь ограничить концепт так, как показано в следующем примере кода, GCC выдает ошибку, что concept cannot be constrained (концепт нельзя ограничивать).

Ограничение концепта

```
template<typename T>
concept AlwaysTrue = true;
```

```
template<typename T>
requires AlwaysTrue<T>
concept Error = true;
```

Рассмотрим концепты Integral, Signed Integral и UnsignedIntegral.

Концепты Integral, Signed Integral и UnsignedIntegral

```
1  template <typename T>
2  concept Integral = std::is_integral<T>::value;
3
4  template <typename T>
5  concept SignedIntegral = Integral<T> && std::is_signed<T>::value;
6
7  template <typename T>
8  concept UnsignedIntegral = Integral<T> && !SignedIntegral<T>;
```

Здесь я использовал признак типа std::is_integral² для определения концепта Integral (строка 2). Благодаря std::is_signed я уточнил концепт Integral до концепта SignedIntegral (строка 4). Наконец, отрицание SignedIntegral дает нам концепт UnsignedIntegral (строка 7).

¹ https://en.cppreference.com/w/cpp/header/type_traits.

² https://en.cppreference.com/w/cpp/types/is_integral.

Проверим их работу.

Использование концептов `Integral`, `Signed Integral` и `UnsignedIntegral`

```
1  // SignedUnsignedIntegrals.cpp
2
3  #include <iostream>
4  #include <type_traits>
5
6  template <typename T>
7  concept Integral = std::is_integral<T>::value;
8
9  template <typename T>
10 concept SignedIntegral = Integral<T> && std::is_signed<T>::value;
11
12 template <typename T>
13 concept UnsignedIntegral = Integral<T> && !SignedIntegral<T>;
14
15 void func(SignedIntegral auto integ) {
16     std::cout << "SignedIntegral: " << integ << '\n';
17 }
18
19 void func(UnsignedIntegral auto integ) {
20     std::cout << "UnsignedIntegral: " << integ << '\n';
21 }
22
23 int main() {
24
25     std::cout << '\n';
26
27     func(-5);
28     func(5u);
29
30     std::cout << '\n';
31
32 }
```

Здесь я использовал сокращенный синтаксис шаблонных функций для перегрузки функции `func` с применением концептов `SignedIntegral` (строка 15) и `UnsignedIntegral` (строка 19). Компилятор сам выбирает подходящую перегруженную функцию:

```
SignedIntegral: -5
UnsignedIntegral: 5
```

Использование концептов SignedIntegral и UnsignedIntegral

Следующий концепт использует дизъюнкцию.

Концепт Arithmetic

```
template<typename T>
concept Arithmetic = std::is_integral<T>::value ||
                    std::is_floating_point<T>::value;
```

4.1.8.2 Выражения requires

Благодаря выражениям `requires` вы можете определять концепты с мощными возможностями. Выражение `requires` имеет следующий вид.

Выражение `requires`

```
requires (parameter-list(optional)) {requirement-seq}
```

- `parameter-list`: разделенный запятыми список параметров, аналогичный списку в определении функции.
- `requirement-seq`: последовательность, состоящая из простых, составных или вложенных требований и требований типа.

4.1.8.2.1 Простые требования

Следующий концепт `Addable` является примером простого требования.

Концепт `Addable`

```
template<typename T>
concept Addable = requires (T a, T b) {
    a + b;
};
```

Концепт `Addable` требует, чтобы было возможно сложение `a + b` двух значений типа `T`.



Избегайте анонимных требований: `requires requires`

Вы можете определить анонимный концепт и сразу же его использовать. Избегайте этого. Это делает ваш код тяжелым для понимания, и вы больше не сможете снова использовать (переиспользовать) эти концепты.

Анонимный концепт для сложения двух концептов

```
template<typename T>
    requires requires (T x) { x + x; }
T add1(T a, T b) { return a + b; }
```

Эта шаблонная функция определяет концепт прямо на месте. Функция `add1` использует выражение `requires` внутри другого выражения `requires`. Используемый анонимный концепт оказывается эквивалентным ранее введенному концепту `Addable`, как и шаблонная функция `add2`, использующая именованный концепт `Addable`.

Использование концепта `Addable`

```
template<Addable T>
T add2(T a, T b) { return a + b; }
```

Концепты должны основываться на общих идеях и давать им понятные имена для переиспользования. Они незаменимы для обеспечения поддержки кода. Анонимные концепты выглядят больше как синтаксические ограничения на параметры шаблона.

4.1.8.2.2 Требования типа

В требованиях типа вы должны использовать ключевое слово `typename` вместе с именем типа.

Концепт `TypeRequirement`

```
template<typename T>
concept TypeRequirement = requires {
    typename T::value_type;
    typename Other<T>;
};
```

Концепт `TypeRequirement` требует, чтобы у типа `T` был вложенный тип `value_type`, а класс `Other` можно было инстанцировать при помощи типа `T`.

Давайте попробуем этот концепт.

Использование концепта `TypeRequirement`

```
1  #include <iostream>
2  #include <vector>
3
4  template <typename>
5  struct Other;
6
7  template <>
8  struct Other<std::vector<int>>> {};
9
```

```

10 template<typename T>
11 concept TypeRequirement = requires {
12     typename T::value_type;
13     typename Other<T>;
14 };
15
16 int main() {
17
18     TypeRequirement auto myVec= std::vector<int>{1, 2, 3};
19
20 }

```

В этом примере выражение `TypeRequirement auto myVec = std::vector<int>{1, 2, 3}` (строка 18) является допустимым. У `std::vector`¹ есть внутренний член `value_type` (строка 12). Шаблонный класс `Other` может быть инстанцирован при помощи `std::vector<int>` (строка 13).

4.1.8.2.3 Составные требования

Составное требование имеет следующий вид.

Составное требование

```
{expression} noexcept(optional) return-type-requirement(optional);
```

Вдобавок к простому требованию составное требование также может содержать спецификатор `noexcept`² и требование на тип возвращаемого значения.

Концепт `Equal`, показанный в следующем примере, использует составные требования.

Определение и использование концепта `Equal`

```

1  // conceptsDefinitionEqual.cpp
2
3  #include <concepts>
4  #include <iostream>
5
6  template<typename T>
7  concept Equal = requires(T a, T b) {
8      { a == b } -> std::convertible_to<bool>;
9      { a != b } -> std::convertible_to<bool>;
10 };

```

¹ <https://en.cppreference.com/w/cpp/container/vector>.

² https://en.cppreference.com/w/cpp/language/noexcept_spec.

```
11
12 bool areEqual(Equal auto a, Equal auto b){
13     return a == b;
14 }
15
16 struct WithoutEqual{
17     bool operator==(const WithoutEqual& other) = delete;
18 };
19
20 struct WithoutUnequal{
21     bool operator!=(const WithoutUnequal& other) = delete;
22 };
23
24 int main() {
25
26     std::cout << std::boolalpha << '\n';
27     std::cout << "areEqual(1, 5): " << areEqual(1, 5) << '\n';
28
29     /*
30
31     bool res = areEqual(WithoutEqual(), WithoutEqual());
32     bool res2 = areEqual(WithoutUnequal(), WithoutUnequal());
33
34     */
35
36     std::cout << '\n';
37
38 }
```

Концепт `Equal` (строка 6) требует, чтобы его параметр `T` поддерживал операторы равенства и неравенства. Кроме того, оба этих оператора должны возвращать значение, конвертируемое в `bool`. Конечно, тип `int` поддерживает концепт `Equal`, но это не относится к типам `WithoutEqual` (строка 16) и `WithoutUnequal` (строка 20). Поэтому когда я использую тип `WithoutEqual` (строка 31), то получаю следующее сообщение об ошибке от компилятора GCC.

```

<source>:6:17:   in requirements with 'T a', 'T b' [with T = WithoutEqual]
<source>:7:9: note: the required expression '(a == b)' is invalid
  7 |         { a == b } -> std::convertible_to<bool>;
    |         ~~~~~
<source>:8:9: note: the required expression '(a != b)' is invalid
  8 |         { a != b } -> std::convertible_to<bool>;
    |         ~~~~~

```

WithoutEqual не удовлетворяет концепту Equal

4.1.8.2.4 Вложенные требования

Вложенные требования имеют следующий вид.

Вложенное требование

```
requires constraint-expression;
```

Вложенные требования используются для задания требований на параметры типа.

Ниже приводится другой способ определения концепта `UnsignedIntegral`.

Концепты `Integral`, `SignedIntegral` и `UnsignedIntegral`

```

1  // nestedRequirements.cpp
2
3  #include <type_traits>
4
5  template <typename T>
6  concept Integral = std::is_integral<T>::value;
7
8  template <typename T>
9  concept SignedIntegral = Integral<T> && std::is_signed<T>::value;
10
11 // template <typename T>
12 // concept UnsignedIntegral = Integral<T> && !SignedIntegral<T>;
13
14 template <typename T>
15 concept UnsignedIntegral = Integral<T> &&
16 requires(T) {
17     requires !SignedIntegral<T>;
18 };
19
20 int main() {
21

```

```

22     UnsignedIntegral auto n = 5u;  // works
23     // UnsignedIntegral auto m = 5;  // compile time error,
24                                     // 5 is a signed literal
25 }

```

В строке 14 используется вложенное требование вместе с концептом `SignedIntegral`, чтобы уточнить концепт `Integral`. Если честно, то закомментированный вариант концепта `UnsignedIntegral` на 11-й строке читать удобнее.

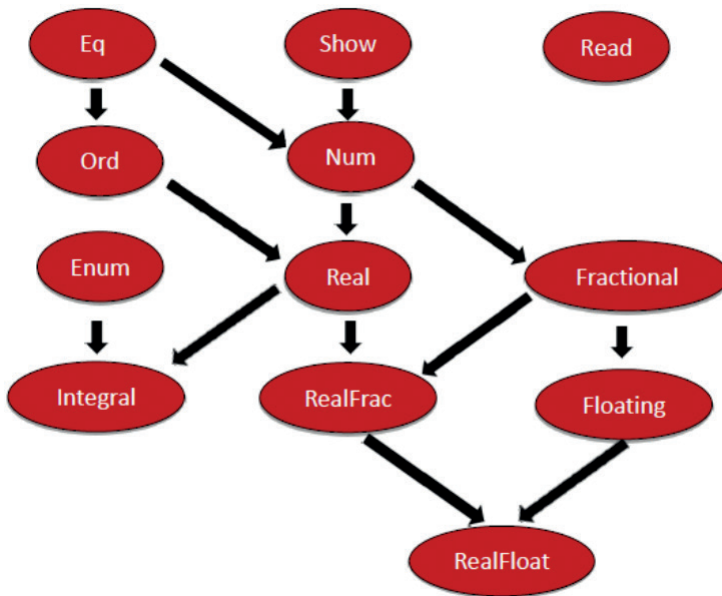
Концепт `Ordering`, вводимый в следующем разделе, демонстрирует применение вложенных требований.

4.1.9 Применение концептов

В предыдущих разделах я ответил на два весьма важных вопроса о концептах «Как можно использовать концепты?» и «Как вы можете определить свой концепт?». В этом разделе я хочу применить теоретические знания из предыдущих разделов для определения более продвинутых концептов, таких как `Ordering`, `SemiRegular` и `Regular`.

4.1.9.1 Концепты `Equal` и `Ordering`

Я уже приводил ранее часть иерархии классов типов в Haskell:



Иерархия классов типов в Haskell

Иерархия показывает, что класс типов `Ord` является уточнением класса `Eq`. В Haskell это выражено очень элегантно.

Часть иерархии классов типов в Haskell

```

1  class Eq a where
2      (==) :: a -> a -> Bool
3      (/=) :: a -> a -> Bool
4
5  class Eq a => Ord a where
6      compare :: a -> a -> Ordering
7      (<) :: a -> a -> Bool
8      (<=) :: a -> a -> Bool
9      (>) :: a -> a -> Bool
10     (>=) :: a -> a -> Bool
11     max :: a -> a -> a

```

Каждый тип `a`, поддерживающий класс `Eq` (строка 1), должен поддерживать равенство (строка 2) и неравенство (строка 3). Далее становится более интересно. Каждый тип `a`, поддерживающий `Ord`, должен поддерживать `Eq` (`class Eq a => Ord a` в строке 5). Кроме того, тип `a` должен поддерживать четыре оператора сравнения и функции `compare` и `max` (строки 6–11).

А теперь давайте посмотрим, сможем ли мы выразить соотношение между `Eq` и `Ord` при помощи концептов C++20. Для простоты я проигнорирую функции `compare` и `max`.

4.1.9.1.1 Концепт `Ordering`

Благодаря выражению `requires` определение концепта `Ordering` выглядит очень похожим на определение `Ord` в Haskell.

Концепт `Ordering`

```

template <typename T>
concept Ordering =
    Equal<T> &&
    requires(T a, T b) {
        { a <= b } -> std::convertible_to<bool>;
        { a < b } -> std::convertible_to<bool>;
        { a > b } -> std::convertible_to<bool>;
        { a >= b } -> std::convertible_to<bool>;
    };

```

Концепт `Ordering` использует вложенные требования внутри себя. Тип `T` поддерживает концепт `Ordering`, если он поддерживает концепт `Equal` и, кроме того, четыре оператора сравнения. Давайте это проверим.

Определение и использование концепта Ordering

```
1  // conceptsDefinitionOrdering.cpp
2
3  #include <concepts>
4  #include <iostream>
5  #include <unordered_set>
6
7  template<typename T>
8  concept Equal =
9      requires(T a, T b) {
10         { a == b } -> std::convertible_to<bool>;
11         { a != b } -> std::convertible_to<bool>;
12     };
13
14
15  template <typename T>
16  concept Ordering =
17      Equal<T> &&
18      requires(T a, T b) {
19         { a <= b } -> std::convertible_to<bool>;
20         { a < b } -> std::convertible_to<bool>;
21         { a > b } -> std::convertible_to<bool>;
22         { a >= b } -> std::convertible_to<bool>;
23     };
24
25  template <Equal T>
26  bool areEqual(const T& a, const T& b) {
27      return a == b;
28  }
29
30  template <Ordering T>
31  T getSmaller(const T& a, const T& b) {
32      return (a < b) ? a : b;
33  }
34
35  int main() {
36
37      std::cout << std::boolalpha << '\n';
38
39      std::cout << "areEqual(1, 5): " << areEqual(1, 5) << '\n';
```

```

40
41     std::cout << "getSmaller(1, 5): " << getSmaller(1, 5) << '\n';
42
43     std::unordered_set<int> firSet{1, 2, 3, 4, 5};
44     std::unordered_set<int> secSet{5, 4, 3, 2, 1};
45
46     std::cout << "areEqual(firSet, secSet): " << areEqual(firSet,
47                                                             secSet) << '\n';
48     // auto smallerSet = getSmaller(firSet, secSet);
49
50     std::cout << '\n';
51
52 }

```

Шаблонная функция `areEqual` (строка 25) требует, чтобы оба аргумента `a` и `b` имели один и тот же тип и поддерживали концепт `Equal`. Дополнительно к этому шаблонная функция `getSmaller` требует, чтобы оба аргумента поддерживали концепт `Ordering`. Конечно, целочисленные значения, такие как 1 и 5, поддерживают оба этих концепта. Тип `std::unordered_set1`, как и следует из его имени, не поддерживает концепт `Ordering`. Поэтому я закомментировал строку 48.

```

areEqual(1, 5): false
getSmaller(1, 5): 1
areEqual(firSet, secSet): true

```

Использование концепта Ordering

Давайте сейчас разберем более интересный случай. Что произойдет, если мы попробуем скомпилировать строку 48: `auto smallerSet = getSmaller(firSet, secSet);`? Компилятор GCC достаточно явно сообщает нам, что `std::unordered_set` не является допустимым аргументом для шаблонной функции `getSmaller`.

```

<source>:48:48:   required from here
<source>:16:9:   required for the satisfaction of 'Ordering<T>' [with T = std::unordered_set<int, std::hash<int>, std::equal_to<int>, std::allocator<int>>]
<source>:18:5:   in requirements with 'T a', 'T b' [with T = std::unordered_set<int, std::hash<int>, std::equal_to<int>, std::allocator<int>>]
<source>:19:13: note: the required expression '(a < b)' is invalid
19 |     { a < b } -> std::convertible_to<bool>;
   |     ~~~~~
<source>:20:13: note: the required expression '(a < b)' is invalid
20 |     { a < b } -> std::convertible_to<bool>;
   |     ~~~~~
<source>:21:13: note: the required expression '(a > b)' is invalid
21 |     { a > b } -> std::convertible_to<bool>;
   |     ~~~~~
<source>:22:13: note: the required expression '(a >= b)' is invalid
22 |     { a >= b } -> std::convertible_to<bool>;
   |     ~~~~~

```

Ошибочное использование функции `getSmaller`

¹ https://en.cppreference.com/w/cpp/container/unordered_set.

На самом деле концепт `Ordering` уже является частью C++20.

- `std::three_way_comparable`: эквивалентно концепту `Ordering`, приведенному выше.
- `std::three_way_comparable_with`: позволяет осуществлять сравнения значений разных типов, например `1.0 < 1.0f`.

В C++20 у нас есть оператор трехстороннего сравнения `<=>`. Более подробно мы рассмотрим его в соответствующей главе.

4.1.9.2 Концепты `SemiRegular` и `Regular`

Когда вы хотите определить конкретный тип, который хорошо работает в экосистеме C++, то вам нужно определить тип, который «ведет себя как `int`». Формально ваш тип должен быть регулярным (`regular`). И в этом разделе я определю типы `SemiRegular` и `Regular`.

Типы `SemiRegular` и `Regular` являются очень важными концептами в C++. Например, вот правило T.46 из «Руководящих принципов C++»: требуйте, чтобы параметры шаблона были как минимум `Regular` или `SemiRegular`¹. Остается один важный вопрос: что это за типы такие, `SemiRegular` и `Regular`? Прежде чем я уйду глубоко в детали, вот неформальный ответ:

- тип `regular` ведет себя как `int`. Он может быть скопирован, и результат операции копирования не зависит от исходного значения и имеет то же самое значение.

Отлично, давайте теперь это формализуем. Также `regular` является `semiregular`, поэтому начнем.



Типы `regular`

Александр Степанов², придумавший идею стандартной библиотеки шаблонов, определил типы `regular` и `semiregular`. Согласно этому, тип является `regular`, если он поддерживает следующие функции:

- создание копии (`copy construction`);
- присваивание;
- равенство;
- уничтожение;
- полное упорядочение.

Создание копии подразумевает создание конструктора по умолчанию (`default construction`), а равенство подразумевает неравенство. Когда Степанов определял эти требования, то в C++ еще не было семантики перемещения. Книга «Элементы программирования»³, которую Александр Степанов написал вместе с Полом Мак Джонсом⁴, посвящена регулярным типам.

¹ <http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-regular>.

² https://en.wikipedia.org/wiki/Alexander_Stepanov.

³ <http://elementsofprogramming.com/>.

⁴ <https://www.mcjones.org/paul/>.

4.1.9.2.1 Концепт SemiRegular

Тип X является `semiRegular`, если он поддерживает большую шестерку функций (Big Six) и операцию `swap`. Большая шестерка состоит из следующих функций:

- конструктор по умолчанию $X()$;
- конструктор копирования $X(\text{const } X\&)$;
- оператор присваивания $X\& \text{ operator }=(\text{const } X\&)$;
- конструктор перемещения $X(X\&\&)$;
- оператор перемещения $X\& \text{ operator }=(X\&\&)$;
- деструктор $\sim X()$.

Дополнительно к этому должна быть определена операция `swap($X\&$, $X\&$)`.

Благодаря библиотеке свойств типов¹ определить соответствующий концепт легко. Для начала я определяю свойство типа `isSemiRegular` и затем использую его для определения концепта `SemiRegular`.

```

1  template<typename T>
2  struct isSemiRegular: std::integral_constant<bool,
3                      std::is_default_constructible<T>::value &&
4                      std::is_copy_constructible<T>::value &&
5                      std::is_copy_assignable<T>::value &&
6                      std::is_move_constructible<T>::value &&
7                      std::is_move_assignable<T>::value &&
8                      std::is_destructible<T>::value &&
9                      std::is_swappable<T>::value >{};
10
11
12 template<typename T>
13 concept SemiRegular = isSemiRegular<T>::value;
```

Свойство типа `isSemiRegular` (строка 1) выполнено, когда выполнены все свойства, соответствующие большой шестерке (строки 3–8), и свойство `std::is_swappable` (строка 9). Все, что после этого остается для определения концепта `SemiRegular`, – это использовать свойство типа `isSemiRegular` (строка 13).

Давайте теперь рассмотрим концепт `Regular`.

4.1.9.2.2 Концепт Regular

Остался еще один шаг, и мы закончим с пояснением концепта `Regular`. В дополнение к свойствам концепта `SemiRegular` концепт `Regular` требует, чтобы в типе было определено сравнение элементов на равенство. Я уже определил концепт `Equal` ранее. Теперь осталось только соединить концепты `Equal` и `SemiRegular` при помощи операции И.

¹ https://en.cppreference.com/w/cpp/header/type_traits.

Определение концепта `Regular`

```
template<typename T>
concept Regular = Equal<T> &&
                  SemiRegular<T>;
```

Теперь давайте рассмотрим, как определены концепты `std::regular` и `std::semi_regular` в C++20.

4.1.9.2.3 `std::semiregular` и `std::regular`

C++20 для построения `std::semiregular` и `std::regular` использует свойства типов и ранее введенные концепты.

Определение концептов `std::semiregular` и `std::regular`

```
template<class T>
concept movable = is_object_v<T> && move_constructible<T> &&
                  assignable_from<T&, T> && swappable<T>;

template<class T>
concept copyable = copy_constructible<T> && movable<T> &&
                  assignable_from<T&, T&> &&
                  assignable_from<T&, const T&> && assignable_from<T&,
                                                                const T>;

template<class T>
concept semiregular = copyable<T> && default_initializable<T>;

template<class T>
concept regular = semiregular<T> && equality_comparable<T>;
```

Интересно то, что концепт `std::regular` определен аналогично `Regular`. С другой стороны, концепт `std::semi_regular` построен из более базовых концептов, таких как `std::copyable` и `std::moveable`. Концепт `std::moveable` основан на свойствах типа `std::is_object`¹. На сайте [cppreference.com](https://en.cppreference.com) также приведена своя реализация предиката, выполняемого во время компиляции.

Возможная реализация свойства типа `std::is_object`

```
template< class T>
struct is_object : std::integral_constant<bool,
    std::is_scalar<T>::value ||
    std::is_array<T>::value ||
    std::is_union<T>::value ||
    std::is_class<T>::value> {};
```

¹ https://en.cppreference.com/w/cpp/types/is_object.

Тип является объектом, если он скаляр, массив, объединение или класс.

В завершение этого раздела я хочу показать применение концептов `Regular` и `std::regular`. Для этого я использую программу `regularSemiRegular.cpp`.

Применение концептов `Regular` и `SemiRegular`

```

1  // regularSemiRegular.cpp
2
3  #include <concepts>
4  #include <vector>
5  #include <type_traits>
6
7  template<typename T>
8  struct isSemiRegular: std::integral_constant<bool,
9                      std::is_default_constructible<T>::value &&
10                     std::is_copy_constructible<T>::value &&
11                     std::is_copy_assignable<T>::value &&
12                     std::is_move_constructible<T>::value &&
13                     std::is_move_assignable<T>::value &&
14                     std::is_destructible<T>::value &&
15                     std::is_swappable<T>::value >{};
16
17  template<typename T>
18  concept SemiRegular = isSemiRegular<T>::value;
19
20  template<typename T>
21  concept Equal =
22      requires(T a, T b) {
23          { a == b } -> std::convertible_to<bool>;
24          { a != b } -> std::convertible_to<bool>;
25      };
26
27  template<typename T>
28  concept Regular = Equal<T> &&
29                  SemiRegular<T>;
30
31  template <Regular T>
32  void behavesLikeAnInt(T) {
33      // ...
34  }
35
```

```
30
31  template <Regular T>
32  void behavesLikeAnInt(T) {
33      // ...
34  }
35
36  template <std::regular T>
37  void behavesLikeAnInt2(T) {
38      // ...
39  }
40
41  struct EqualityComparable { };
42  bool operator == (EqualityComparable const&,
43                  EqualityComparable const&) {
44      return true;
45  }
46
47  struct NotEqualityComparable { };
48
49  int main() {
50
51      int myInt{};
52      behavesLikeAnInt(myInt);
53      behavesLikeAnInt2(myInt);
54
55      std::vector<int> myVec{};
56      behavesLikeAnInt(myVec);
57      behavesLikeAnInt2(myVec);
58
59      EqualityComparable equComp;
60      behavesLikeAnInt(equComp);
61      behavesLikeAnInt2(equComp);
62
63      NotEqualityComparable notEquComp;
64      behavesLikeAnInt(notEquComp);
65      behavesLikeAnInt2(notEquComp);
66
67  }
```

Я собрал здесь вместе все фрагменты из предыдущих примеров кода для определения концепта `Regular` (строка 27). Шаблонная функция `behaves-LikeAnInt` (строка 36) проверяет, ведет ли себя ее аргумент как `int`. Это значит, что определенный пользователем концепт `Regular` и концепт из C++20 `std::regular` применяются для проверки условия. Как подсказывает название, тип `EqualityComparable` (строка 41) поддерживает проверку на равенство, но тип `NotEqualityComparable` (строка 47) – нет. Использование типа `NotEqualityComparable` в обоих вызовах функций (строки 64 и 65) является наиболее интересной частью этой программы.

Хотя я пока нахожусь на ранних стадиях реализации концептов, я хочу сравнить сообщения об ошибках от компиляторов GCC и MSVC.

○ GCC

Я использовал GCC 10.2 с параметром `-std=c++20` на сайте [Compiler Explorer](https://godbolt.org/)¹. Вот сообщения об ошибках, которые я получил при использовании концепта `Regular` (строка 64):

```
<source>:23:13: note: the required expression 'a == b' is invalid
 23 |         { a == b } -> std::convertible_to<bool>;
    |         ~~~~~
<source>:24:13: note: the required expression 'a != b' is invalid
 24 |         { a != b } -> std::convertible_to<bool>;
    |         ~~~~~
```

Сообщения об ошибках при использовании концепта `Regular`

В случае концепта C++20 `std::regular` мы получаем более содержательные сообщения об ошибках. Строка 65 – более информативное сообщение об ошибке:

```
/opt/compiler-explorer/gcc-10.2.0/include/c++/10.2.0/concepts:282:10: note: the required expression '\_\_t == \_\_u' is invalid
282 |     { \_\_t == \_\_u } -> __boolean_testable;
    |     ~~~~~
/opt/compiler-explorer/gcc-10.2.0/include/c++/10.2.0/concepts:283:10: note: the required expression '\_\_t != \_\_u' is invalid
283 |     { \_\_t != \_\_u } -> __boolean_testable;
    |     ~~~~~
/opt/compiler-explorer/gcc-10.2.0/include/c++/10.2.0/concepts:284:10: note: the required expression '\_\_u == \_\_t' is invalid
284 |     { \_\_u == \_\_t } -> __boolean_testable;
    |     ~~~~~
/opt/compiler-explorer/gcc-10.2.0/include/c++/10.2.0/concepts:285:10: note: the required expression '\_\_u != \_\_t' is invalid
285 |     { \_\_u != \_\_t } -> __boolean_testable;
    |     ~~~~~
```

Сообщения об ошибках при использовании концепта `std::regular`

○ MSVC

Сообщение об ошибке, выданное MSVC, слишком неконкретное.

¹ <https://godbolt.org/>.

```

C:\Users\seminar>cl.exe /EHsc /std:c++latest regularSemiRegular.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 19.27.29112 for x64
Copyright (c) Microsoft Corporation. All rights reserved.

/std:c++latest is provided as a preview of language features from the latest C++
working draft, and we're eager to hear about bugs and suggestions for improvements.
However, note that these features are provided as-is without support, and subject
to changes or removal as the working draft evolves. See
https://go.microsoft.com/fwlink/?linkid=2045807 for details.

regularSemiRegular.cpp
regularSemiRegular.cpp(64): error C2672: 'behavesLikeAnInt': no matching overloaded function found
regularSemiRegular.cpp(64): error C7602: 'behavesLikeAnInt': the associated constraints are not satisfied
regularSemiRegular.cpp(32): note: see declaration of 'behavesLikeAnInt'
regularSemiRegular.cpp(65): error C2672: 'behavesLikeAnInt2': no matching overloaded function found
regularSemiRegular.cpp(65): error C7602: 'behavesLikeAnInt2': the associated constraints are not satisfied
regularSemiRegular.cpp(37): note: see declaration of 'behavesLikeAnInt2'

C:\Users\seminar>

```

Сообщение об ошибках, выданное при использовании концептов Regular и `std::regular`

На скриншоте показано, что была использована сборка 19.27.29112 для x64 со следующей командной строкой `/EHSC /std:c++latest`.



Концепты в C++20: эволюция или революция?

Следующий небольшой обзор выражает мое личное мнение. Сначала я приведу факты, а потом сделаю из них выводы. Факты основаны на том, что я представил в данной главе. Итак, какие аргументы говорят за эволюцию и какие – за революцию?

Эволюция

- ♦ Концепты продвигают работу с обобщенным кодом на **более высоком уровне абстракции**.
- ♦ Концепты дают вам понятные сообщения об ошибках, когда компиляция шаблона приводит к ошибке. То, что они дают, нельзя достичь при помощи библиотеки свойств типов¹, SFINAE² и `static_assert`³.
- ♦ `auto` выступает в качестве заполнителя без каких-либо ограничений. С C++20 вы можете использовать концепты как **заполнители с ограничением**.
- ♦ Начиная со стандарта C++14 вы можете использовать **обобщенные лямбды** как удобный способ задания шаблонных функций.

Революция

- ♦ Концепты позволяют нам впервые **проверять требования шаблона**. Конечно, вы также могли достичь верификации параметров шаблона при помощи комбинации библиоте-

¹ https://en.cppreference.com/w/cpp/header/type_traits.

² <https://en.cppreference.com/w/cpp/language/sfinae>.

³ https://en.cppreference.com/w/cpp/language/static_assert.

ки свойств типов¹, SFINAE² и `static_assert`³, но это слишком сложно в качестве общего решения.

- ♦ Благодаря сокращенному синтаксису шаблонных функций задание шаблонов стало гораздо проще.
- ♦ Концепты представляют собой **семантические категории**, а не синтаксические ограничения. Вместо такого концепта, как `Addable`, требующего поддержки оператора `+`, мы должны думать в категориях `Number`, где `Number` – это семантическая категория, такая как `Equal` или `Ordering`.

Выводы

Есть много споров о том, являются концепты эволюционным шагом или революционным прыжком вперед. В основном из-за семантических категорий я на стороне революционного прыжка. Такие концепты, как `Number`, `Equality` или `Ordering`, напоминают мне мир идей Платона⁴. *Революционно то, как мы теперь можем думать о программировании в терминах этих категорий.*



Важные замечания

- ♦ Использование функций или классов, определенных для заданного типа данных, приводит к возникновению ряда проблем. Концепты преодолевают эти проблемы, позволяя задавать семантические ограничения на параметры.
- ♦ Концепты могут быть использованы в директивах `require` как параметры шаблона с ограничением или же в сокращенной форме шаблонных функций.
- ♦ Концепты являются предикатами времени компиляции и могут быть использованы в самых разных шаблонах. Вы можете осуществлять перегрузку при помощи концептов, специализировать шаблоны при помощи концептов, использовать концепты для методов класса или шаблонов с переменным количеством аргументов.
- ♦ Благодаря C++20 и концептам использование заполнителей без ограничений (`auto`) и заполнителей с ограничением (концепты) было унифицировано. Везде, где вы применяете `auto`, вы можете использовать концепты.
- ♦ Благодаря новому сокращенному синтаксису для задания шаблонов задание шаблонной функции стало очень простым и красивым.
- ♦ Не изобретайте колесо. Прежде чем определять свои собственные концепты, изучите богатый набор предопределенных концептов в стандарте C++20. При определении своих собственных концептов вы можете применять два приема: комбинировать концепты и предикаты времени компиляции и использовать директиву `requires`.

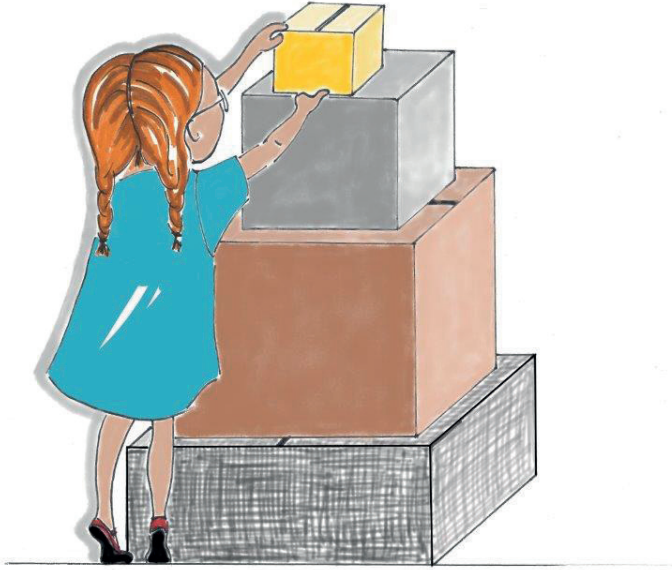
¹ https://en.cppreference.com/w/cpp/header/type_traits.

² <https://en.cppreference.com/w/cpp/language/sfinae>.

³ https://en.cppreference.com/w/cpp/language/static_assert.

⁴ <https://en.wikipedia.org/wiki/Plato>.

4.2 Модули



Сиппи подготавливает пакеты (packages)

Модули являются одной из четырех больших ключевых возможностей C++20: концепты, модули, диапазоны и корутины. Модули много чего обещают: уменьшенное время компиляции, изоляция макросов, уход от заголовочных файлов и разных костылей. Прежде чем я перейду к преимуществам модулей, хочу отступить назад и объяснить их преимущества.

4.2.1 Для чего нужны модули?

Давайте начнем с простого примера. Рассмотрим программу `helloWorld.cpp`.

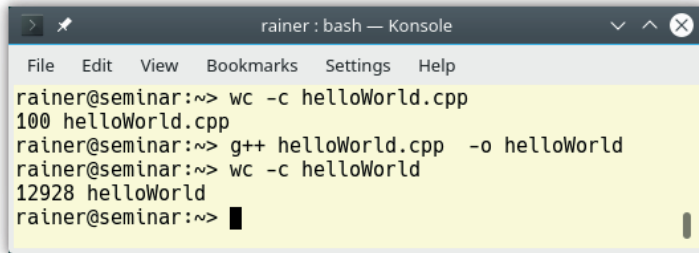
Простая программа `hello world`

```
// helloWorld.cpp

#include <iostream>

int main() {
    std::cout << "Hello World" << '\n';
}
```

Создание исполняемого файла `helloWorld` из этого файла при помощи GCC¹ увеличивает его размер почти в 130 раз.



```
rainer : bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~> wc -c helloWorld.cpp
100 helloWorld.cpp
rainer@seminar:~> g++ helloWorld.cpp -o helloWorld
rainer@seminar:~> wc -c helloWorld
12928 helloWorld
rainer@seminar:~>
```

Размер объектного файла

Числа 100 и 12928 на приведенном скриншоте соответствуют количеству занимаемых байтов. Отлично. Теперь у нас будет базовое понимание того, что происходит.

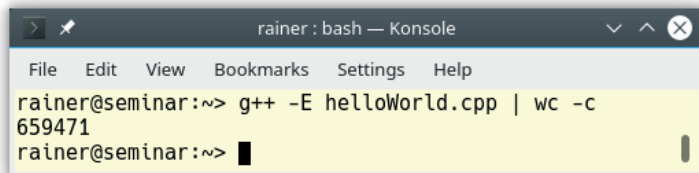
4.2.1.1 Классический процесс сборки

Процесс сборки состоит из трех шагов: препроцессинг, компиляция и линковка.

4.2.1.1.1 Препроцессинг

Препроцессор обрабатывает такие директивы, как `#include` и `#define`. Препроцессор заменяет `#include` содержимым соответствующих заголовочных файлов и заменяет макросы (`#define`). Благодаря таким директивам, как `#if`, `#else`, `#elif`, `#ifdef`, `#ifndef` и `#endif`, части исходного кода могут быть включены или исключены из процесса компиляции.

Этот прямолинейный процесс подстановки текста может наблюдаться при помощи установки флага компилятору `-E` в GCC/Clang или `/E` Windows.



```
rainer : bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~> g++ -E helloWorld.cpp | wc -c
659471
rainer@seminar:~>
```

Вывод препроцессора

Можно заметить, что выходной файл после препроцессинга содержит более полумиллиона байт. Я не хочу обвинять GCC, другие компиляторы ведут себя примерно так же. Выход препроцессора является входом для компилятора.

Результатом этапа препроцессинга является объект трансляции (translation unit).

¹ <http://gcc.gnu.org/>.

4.2.1.1.2 Компиляция

Компиляция выполняется отдельно для каждого выхода препроцессора. Компилятор разбирает исходный код на C++ и переводит его в код на ассемблере. Созданный файл называется объектным и содержит откомпилированный код в бинарной форме. Объектный код может ссылаться на символы, которые не были определены. Объектные файлы могут помещаться в архивы для дальнейшего переиспользования. Такие архивы называются статическими библиотеками.

Созданные компилятором объектные файлы являются входом для линковки (сборки).

4.2.1.1.3 Линковка

Результатом работы компоновщика (линковщика) может быть исполняемый файл, статическая или динамическая библиотека. Задача линкера – разрешить все ссылки к неопределенным символам. Символы определены в объектных файлах и библиотеках. Типичная ошибка на этой стадии – какие-то символы не определены или определены более одного раза.

Этот процесс сборки, состоящий из трех шагов, унаследован из C. Он работает достаточно хорошо, если у вас только одна единица трансляции (translation unit). Но когда их у вас много, может возникнуть множество проблем.

4.2.1.2 Проблемы в процессе сборки

Ниже приводится неполный список возможных проблем в классическом процессе сборки, которые можно обойти при помощи модулей.

4.2.1.2.1 Повторяемая подстановка

Препроцессор заменяет директивы `#include` соответствующими заголовочными файлами. Давайте изменим изначальный текст программы `helloWorld.cpp` так, чтобы повторение стало видным.

Я переработал программу и добавил два исходных файла `hello.cpp` и `world.cpp`. Исходный файл `hello.cpp` содержит функцию `hello()`, исходный файл `world.cpp` содержит функцию `world()`. Оба исходных файла подключают соответствующие заголовочные файлы. Переработка привела к тому, что теперь у программы есть некоторое внешнее поведение, как и в `helloWorld.cpp`, но внутренняя структура улучшена. Вот новые файлы:

- `hello.cpp` и `hello.h`:

Реализация `hello`

```
// hello.cpp
```

```
#include "hello.h"
```

```
void hello() {  
    std::cout << "hello ";  
}
```

Заголовочный файл к hello

```
// hello.h
```

```
#include <iostream>
```

```
void hello();
```

○ world.cpp и world.h:

Реализация world

```
// world.cpp
```

```
#include "world.h"
```

```
void world() {  
    std::cout << "world";  
}
```

Заголовочный файл к world

```
// world.h
```

```
#include <iostream>
```

```
void world();
```

○ helloWorld2.cpp:

Использование hello и world

```
// helloWorld2.cpp
```

```
#include <iostream>
```

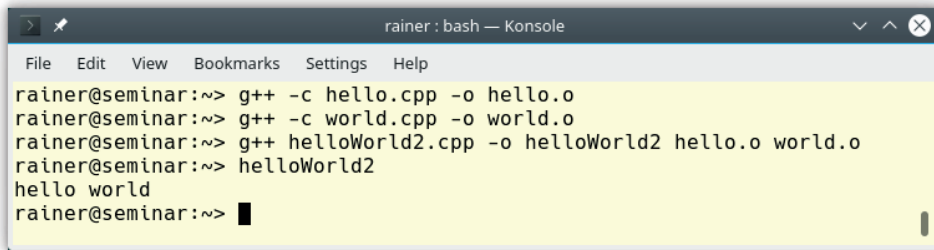
```
#include "hello.h"
```

```
#include "world.h"
```

```
int main() {  
    hello();  
    world();  
    std::cout << '\n';
```

```
}
```

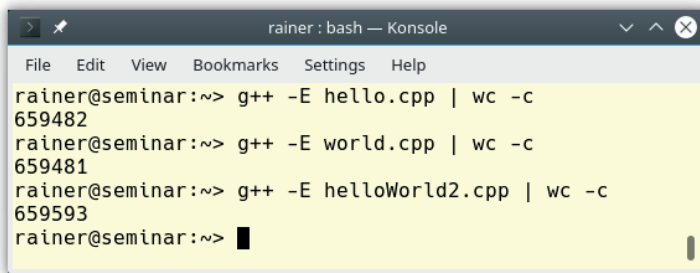
Сборка и выполнение программы работает, как и ожидалось:



```
rainer : bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~> g++ -c hello.cpp -o hello.o
rainer@seminar:~> g++ -c world.cpp -o world.o
rainer@seminar:~> g++ helloWorld2.cpp -o helloWorld2 hello.o world.o
rainer@seminar:~> helloWorld2
hello world
rainer@seminar:~> █
```

Компиляция простой программы

Но существует проблема. Препроцессор выполняется для каждого входного файла. Это значит, что заголовочный файл `<iostream>` включается три раза. Соответственно, каждый файл увеличивается почти на полмиллиона байт.



```
rainer : bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~> g++ -E hello.cpp | wc -c
659482
rainer@seminar:~> g++ -E world.cpp | wc -c
659481
rainer@seminar:~> g++ -E helloWorld2.cpp | wc -c
659593
rainer@seminar:~> █
```

Размер препроцессированного исходного файла

Это трата времени компиляции.

В отличие от заголовочных файлов, модуль импортируется всего один раз и фактически без затрат времени.

4.2.1.2.2 Изоляция от макросов препроцессора

Если и есть согласие с чем-то в сообществе C++, то это следующее: мы должны избавиться от макросов препроцессора. Почему? Использование макроса – это просто подстановка текста, игнорирующая семантику C++. И конечно, у этого есть много негативных последствий: например, результат может зависеть от того, в каком порядке вы включаете макросы, или макрос может конфликтовать с уже определенным макросом либо именем в вашей программе.

Представьте, что у вас есть два заголовочных файла `webcolors.h` и `productinfo.h`.

Первое определение макроса `RED`

```
// webcolors.h
```

```
#define RED    0xFF0000
```

Второе определение макроса RED

```
// productinfo.h
```

```
#define RED 0
```

Теперь, когда исходный файл `client.cpp` включает оба этих заголовочных файла, значение макроса RED зависит от того, в каком порядке эти файлы были включены. Подобное поведение легко приводит к ошибкам.

С модулями порядок включения не играет никакой роли.

4.2.1.2.3 Множественные определения символов

Есть такая аббревиатура ODR – One Definition Rule (правило одного определения), и по поводу функций она говорит следующее:

- у функции не может быть более одного определения в любой единице компиляции;
- у функции не может быть более одного определения в программе.

Встроенные функции (inline functions) с внешней линковкой могут быть определены больше чем в одной единице компиляции. Определения должны быть одинаковыми.

Давайте посмотрим, что мой линкер скажет, когда я попытаюсь слинковать программу, нарушающую правило ODR. В следующем примере у нас будет два заголовочных файла – `header.h` и `header2.h`. Главная программа включает файл `header.h` дважды и тем самым нарушает правило ODR, поскольку будут присутствовать сразу два определения `func`.

Определение функции `func`

```
// header.h
```

```
void func() {}
```

Неявное включение определения функции

```
// header2.h
```

```
#include "header.h"
```

Двойное определение функции `func`

```
// main.cpp
```

```
#include "header.h"
```

```
#include "header2.h"
```

```
int main() {}
```

В результате линкер пожалуется на многочисленное определение func:

```

rainer@seminar:~> g++ main.cpp
In file included from header2.h:3:0,
                  from main.cpp:4:
header.h: In function 'void func()':
header.h:3:6: error: redefinition of 'void func()'
void func(){}
    ^~~~~
In file included from main.cpp:3:0:
header.h:3:6: note: 'void func()' previously defined here
void func(){}
    ^~~~~
rainer@seminar:~>

```

Нарушение ODR

Здесь мы использовали довольно грязный трюк включения одного заголовочного файла в другой. Это можно исправить следующим образом при помощи макроса FUNC_H.

Решение проблемы с нарушением ODR

```

// header.h

#ifdef FUNC_H
#define FUNC_H

void func(){}

#endif

```

С модулями крайне маловероятно встретить дублирующее определение функции.

Теперь подведем итоги рассмотрения преимуществ использования модулей.

4.2.2 Преимущества использования модулей

Вот краткий список преимуществ использования модулей:

- Модули импортируются всего один раз и практически бесплатно.
- Не важно, в каком порядке вы включаете модули.
- Крайне маловероятно столкнуться с дублирующими символами.
- Модули позволяют вам выражать логическую структуру вашего кода. Вы можете явно задавать имена, которые необходимо экспортировать или не нужно экспортировать. Кроме того, вы можете объединить не-

сколько модулей в большой модуль и предоставить вашему клиенту как логический пакет.

- Благодаря модулям нет необходимости разделять исходный код на интерфейс и реализацию.



Длинная история

Модули в C++ могут быть старше, чем вы думали. Далее я приведу краткий исторический обзор, чтобы дать вам понимание, сколько времени нужно, чтобы привести что-то столь важное в стандарт C++.

В 2004 году Давид Вандевоорд (Daveed Vandevoord) написал предложение T1736.pdf¹, в котором впервые была описана идея модулей. Потребовалось время до 2012 года, чтобы создать специальную группу SG2 Modules. В 2017 году Clang 5.0 и MSVC 19.1 представили первые реализации. Годом позже была завершена подготовка Modules TS (technical specification). Примерно в это же время Google предложил так называемый ATOM (Another Take On Modules) (P0947²) для модулей. В 2019 году Modules TS и ATOM были объединены в черновике стандарта C++20 (N4842³).

4.2.3 Простой пример использования модулей

Цель данного раздела проста: дать введение в модули. Более продвинутые возможности модулей будут рассмотрены в последующих разделах. Давайте начнем с простого модуля `math`.

Простой модуль `math`

```
// math.ixx
```

```
export module math;
```

```
export int add(int fir, int sec){
    return fir + sec;
}
```

Выражение `export module math` – это объявление модуля. Помещая слово `export` перед объявлением функции `add`, мы экспортируем `add`, делая ее доступной потребителям нашего модуля.

¹ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1736.pdf>.

² <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0947r1.html>.

³ <https://github.com/cplusplus/draft/releases/tag/n4842>.

Использование модуля `math`

```
// client.cpp
```

```
import math;

int main() {

    add(2000, 20);

}
```

Конструкция `import math` импортирует модуль `math` и делает экспортированные имена из этого модуля видимыми в `client.cpp`.

Давайте начнем с файла объявления модуля (module declaration file).

4.2.3.1 Файл объявления модуля

Заметили ли вы странное имя модуля: `math.ixx`?

- Компилятор от Microsoft использует расширение `ixx`. Суффикс `ixx` обозначает исходный файл интерфейса модуля.
- Компилятор Clang изначально использовал расширение `srml`. Суффикс `m`, вероятно, обозначал модуль. Но позже это соглашение было изменено в новых версиях Clang на расширение `cpp`.
- Компилятор GCC не использует какого-либо специального расширения.

Глобальная часть модуля задает интерфейс модуля. Она начинается с ключевого слова `module` и заканчивается описанием модуля. Это то место, где можно использовать такие директивы, как `#include`, чтобы интерфейс модуля мог быть откомпилирован. Этот код не экспортируется интерфейсом модуля.

Вторая версия модуля `math` поддерживает две функции – `add` и `getProduct`.

Определение модуля с глобальной частью

```
1 // math1.ixx
2
3 module;
4
5 #include <numeric>
6 #include <vector>
7
8 export module math;
9
10 export int add(int fir, int sec){
11     return fir + sec;
12 }
```

```

13
14 export int getProduct(const std::vector<int>& vec) {
15     return std::accumulate(vec.begin(), vec.end(),
                             1, std::multiplies<int>());
16 }

```

Я включил все необходимые заголовочные файлы перед глобальной частью (строка 3) и определением модуля (строка 8).

Использование улучшенного модуля `math`

// client1.cpp

```

#include <iostream>
#include <vector>

import math;

int main() {

    std::cout << '\n';

    std::cout << "add(2000, 20): " << add(2000, 20) << '\n';

    std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

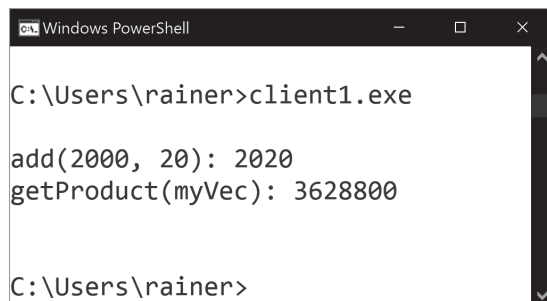
    std::cout << "getProduct(myVec): " << getProduct(myVec) << '\n';

    std::cout << '\n';

}

```

Клиентская часть импортирует модуль `math` и использует его:



```

C:\Users\rainer>client1.exe

add(2000, 20): 2020
getProduct(myVec): 3628800

C:\Users\rainer>

```

Выполнение программы `client1.exe`

Теперь давайте перейдем к деталям.

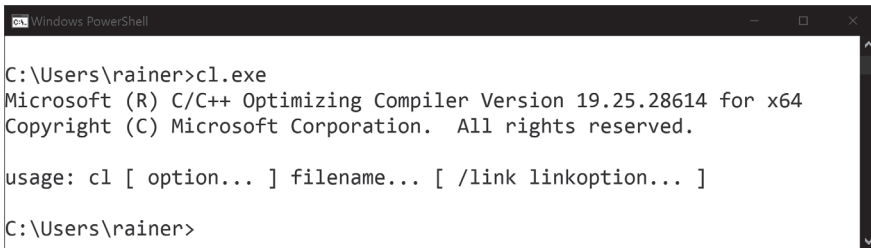
4.2.4 Компиляция и использование

Для компиляции модуля `math.ixx`, используемого клиентской программой `client.cpp`, вы должны воспользоваться свежей версией Clang, GCC или компилятором от Microsoft.

Компиляция модуля является сложной задачей. Именно поэтому я покажу компиляцию примера при помощи Clang и компилятора от Microsoft.

4.2.4.1 Компилятор от Microsoft Visual Studio

Я буду использовать x64 компилятор `cl.exe` 19.25.28614.



```
C:\Users\rainer>cl.exe
Microsoft (R) C/C++ Optimizing Compiler Version 19.25.28614 for x64
Copyright (C) Microsoft Corporation. All rights reserved.

usage: cl [ option... ] filename... [ /link linkoption... ]

C:\Users\rainer>
```

Использование компилятора от Microsoft для модулей

Для компиляции и использования модуля компилятором от Microsoft нужны следующие шаги. Я приведу минимальную командную строку. Кроме того, при применении более старого компилятора от Microsoft вам нужно будет использовать флаг `/std::cpplatest`.

Построение выполняемого файла при помощи компилятора от Microsoft

-
- ```
1 cl.exe /experimental:module /c math.ixx
2 cl.exe /experimental:module client.cpp math.obj
```
- 

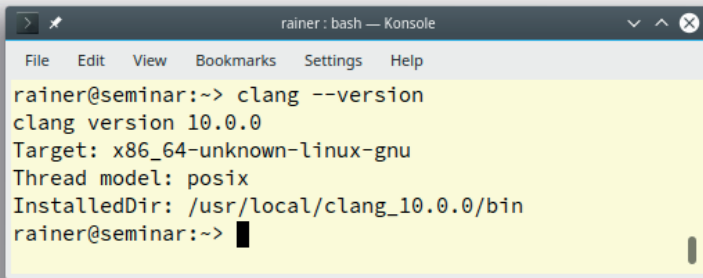
- Строка 1 создает файл `math.obj` и IFC-файл `math.ifc`. IFC-файл содержит метаданные, описывающие интерфейс модуля. Бинарный формат IFC был создан на основе работы Internal Format Representation<sup>1</sup> Габриеля Дос Рейса и Бьярна Страуструпа (2004–2005).
- Строка 2 создает исполняемый файл `client.exe`. Без неявно используемого файла `math.ifc` с предыдущего шага линкер не смог бы найти наш модуль.

##### 4.2.4.2 Компилятор Clang

Под Linux я использую компилятор Clang 10.0.0.

---

<sup>1</sup> <https://www.stroustrup.com/gdr-bs-macis09.pdf>.



```
rainer@seminar:~> clang --version
clang version 10.0.0
Target: x86_64-unknown-linux-gnu
Thread model: posix
InstalledDir: /usr/local/clang_10.0.0/bin
rainer@seminar:~>
```

Компилятор Clang для модулей

При использовании компилятора Clang файл описания модуля – это просто `cpp`-файл. Поэтому я должен переименовать `math.ixx` в `math.cpp`.

Простой модуль `math`

---

```
// math.cpp
```

```
export module math;
```

```
export int add(int fir, int sec){
 return fir + sec;
}
```

---

Клиентский файл `client.cpp` не нужно менять. Ниже приводятся шаги, необходимые для создания исполнимого файла.

Построение исполнимого файла при помощи компилятора Clang

---

```
1 clang++ -std=c++2a -stdlib=libc++ -c math.cpp -Xclang \
2 -emit-module-interface -o math.pcm
3
4 clang++ -std=c++2a -stdlib=libc++ -fprebuilt-module-path= \
5 . client.cpp math.pcm -o client
```

---

- Строка 1 создает модуль `math.pcm`. Суффикс `pcm` обозначает `precompiled module` (предварительно откомпилированный модуль). Флаги `-std=c++2a` задают использование стандарта C++20, `-stdlib=libc++` задают используемую C++ стандартную библиотеку. Флаги `-Xclang -emit-module-interface` необходимы для создания предварительно откомпилированного модуля (`precompiled module`).
- Строка 4 создает исполнимый файл `client`, который использует модуль `math.pcm`. Вы задаете путь к модулю при помощи флага `-fprebuilt-module-path`.

#### 4.2.4.3 Используемый компилятор

В этой книге я использую `cl.exe` от Microsoft. На данный момент (конец 2020 года) у Microsoft наилучшая поддержка модулей<sup>1</sup>. Блог от Microsoft содержит два великолепных текста по модулям – *Overview of modules in C++<sup>2</sup>* и *C++ Modules conformance improvements with MSVC in Visual Studio 2019 16.5<sup>3</sup>*. Ни для Clang, ни для GCC нет подобных поясняющих статей, что делает довольно затруднительным использование модулей с этими компиляторами.

#### 4.2.5 Экспорт из модуля

Есть три способа для экспорта имен с части интерфейса модуля.

##### 4.2.5.1 Спецификатор `export`

Вы можете явно экспортировать имена.

Спецификатор `export`

---

```
export module math;
```

```
export int mult(int fir, int sec);
```

```
export void doTheMath();
```

---

##### 4.2.5.2 Групповой экспорт

Групповой экспорт экспортирует все имена внутри операторных скобок.

Групповой экспорт

---

```
export module math;
```

```
export {
```

```
 int mult(int fir, int sec);
```

```
 void doTheMath();
```

```
}
```

---

##### 4.2.5.3 Экспорт пространства имен

Вместо группового экспорта можно использовать экспорт пространства имен (*exported namespace*).

---

<sup>1</sup> [https://en.cppreference.com/w/cpp/compiler\\_support](https://en.cppreference.com/w/cpp/compiler_support).

<sup>2</sup> <https://docs.microsoft.com/en-us/cpp/cpp/modules-cpp?view=msvc-160&viewFallbackFrom=vs-2019>.

<sup>3</sup> <https://devblogs.microsoft.com/cppblog/c-modules-conformance-improvements-with-msvc-in-visual-studio-2019-16-5/>.

---

Экспорт пространства имен

---

```
export module math;

export namespace math {

 int mult(int fir, int sec);
 void doTheMath();

}
```

---

Когда клиент использует имена из экспортируемого пространства имен, необходимо должным образом задавать эти имена.

Только имена, у которых нет внутренней линковки, могут быть экспортированы.

## 4.2.6 Рекомендации по структуре модуля

Давайте рассмотрим, как структурировать модуль.

Рекомендации по структуре модуля

---

```
module; // global module fragment

#include <headers for libraries not modularized so far>

export module math; // module declaration; starts the module purview

import <importing of other modules>

<non-exported declarations> // names only visibile inside the module

export namespace math {

 <exported declarations> // exported names

}
```

---

Рекомендации служат одной цели: дать вам понимание упрощенной структуры модуля и представление о том, что я собираюсь описывать дальше. Итак, что нового в структуре модуля?

- Глобальная часть модуля, начинающаяся с ключевого слова `module`, не обязательна. После нее и до описания модуля находится место для подключения заголовочных файлов.
- Описание модуля `export module math` начинает область видимости модуля, который заканчивается в конце единицы компиляции.

- Вы можете импортировать модули в начале области видимости модуля. Импортируемые модули имеют линковку на уровне модуля и не видны вне этого модуля. Это также относится к неэкспортируемым объявлениям.
- Я помещаю экспортируемые имена в пространство имен `math`, которое имеет то же имя, что и сам модуль.
- Модуль содержит только объявленные имена. Давайте рассмотрим разделение интерфейса и реализации модуля.

## 4.2.7 Блок интерфейса модуля и блок реализации модуля

Когда ваш модуль становится больше, вам следует структурировать его в блок интерфейса модуля и в один или несколько блоков реализации. Следуя ранее упомянутым рекомендациям, я переделаю предыдущую версию модуля `math`.

### 4.2.7.1 Блок интерфейса модуля

Блок интерфейса модуля

---

```
1 // mathInterfaceUnit.ixx
2
3 module;
4
5 #include <vector>
6
7 export module math;
8
9 export namespace math {
10
11 int add(int fir, int sec);
12
13 int getProduct(const std::vector<int>& vec);
14
15 }
```

---

- Блок интерфейса модуля содержит описания экспортируемых объявлений: `export module math` (строка 7).
- Имена `add` и `getProduct` экспортируются (строки 11 и 13).
- У модуля может быть только один блок интерфейса.

#### 4.2.7.2 Блок реализации модуля

Блок реализации модуля

---

```

1 // mathImplementationUnit.cpp
2
3 module math;
4
5 #include <numeric>
6
7 namespace math {
8
9 int add(int fir, int sec) {
10 return fir + sec;
11 }
12
13 int getProduct(const std::vector<int>& vec) {
14 return std::accumulate(vec.begin(), vec.end(),
15 1, std::multiplies<int>());
16 }
17 }
```

---

- Реализация модуля содержит неэкспортируемые описания: `module math` (строка 3).
- У модуля может быть более чем один блок реализации модуля.

#### 4.2.7.3 Основная программа

Использование модуля `math`

---

```

1 // client3.cpp
2
3 #include <iostream>
4 #include <vector>
5
6 import math;
7
8 int main() {
9
10 std::cout << '\n';
11
12 std::cout << "math::add(2000, 20): " << math::add(2000, 20) << '\n';
13 }
```

---

```
14 std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
15
16 std::cout << "math::getProduct(myVec): " << math:
 :getProduct(myVec) << '\n';
17
18 std::cout << '\n';
19
20 }
```

---

С точки зрения пользователя модуль `math` подключен, а пространство имен `math` было добавлено в строке 6.

Когда мои объяснения начинают зависеть от используемого компилятора, я помещаю их в отдельный блок в книге. Эта информация крайне важна в случае, если вы хотите попробовать выполнить код сами.



### Сборка исполняемого файла при помощи компилятора от Microsoft

Ручная сборка исполняемого файла состоит из нескольких шагов.

Построение модуля с блоком интерфейса и блоком реализации

---

```
1 cl.exe /c /experimental:module mathInterfaceUnit.ixx /EHsc
2 cl.exe /c /experimental:module mathImplementationUnit.cpp /EHsc
3 cl.exe /c /experimental:module client3.cpp /EHsc
4 cl.exe client3.obj mathInterfaceUnit.obj mathImplementationUnit.obj
```

---

- Строка 1 создает объектный файл `mainInterfaceUnit.obj` и модуль интерфейса `math.ifc`.
- Строка 2 создает объектный файл `mathImplementationUnit.obj`.
- Строка 3 создает объектный файл `client3.obj`.
- Строка 4 создает выполнимый файл `client3.exe`.

При использовании компилятора от Microsoft вам необходимо указать модель обработки исключений (`/EHsc`) и включить поддержку модулей (`/experimental:module`).

Результат выполнения программы следующий:

```
C:\Users\rainer>client3

math::add(2000, 20): 2020
math::getProduct(myVec): 3628800

C:\Users\rainer>
```

Выполнение программы `client2.exe`



## 4.2.8 Подмодули и разделы модулей

Когда ваш модуль становится больше, вы можете захотеть разделить его функциональность на отдельные компоненты. C++20 предоставляет для этого два подхода: подмодули (submodules) и разделы модуля (partitions).

### 4.2.8.1 Подмодули

Модуль может импортировать другие модули и затем реэкспортировать их.

В следующем примере модуль `math` импортирует подмодули `math.math1` и `math.math2`.

Модуль `math`

---

```
// mathModule.ixx
```

```
export module math;
```

```
export import math.math1;
```

```
export import math.math2;
```

---

Выражение `export import math.math1` импортирует модуль `math.math1` и затем экспортирует его как часть модуля `math`.

Для полноты ниже приводятся модули `math.math1` и `math.math2`. Я использую точку для разделения модуля `math` от подмодулей. Эта точка не обязательна.

Подмодуль `math.math1`

---

```
// mathModule1.ixx
```

```
export module math.math1;
```

```
export int add(int fir, int sec) {
 return fir + sec;
}
```

---

Подмодуль `math.math2`

---

```
// mathModule2.ixx
```

```
export module math.math2;
```

```
export {
 int mul(int fir, int sec) {
 return fir * sec;
 }
}
```

---

Если присмотреться к коду, то можно заметить небольшую разницу в операторе `export` в модуле `math`. В то время как `math.math1` использует спецификатор `export`, `math.math2` использует групповой экспорт.

С точки зрения программиста использование модуля `math` очень просто.

Главная программа

---

```
// mathModuleClient.cpp

#include <iostream>

import math;

int main() {

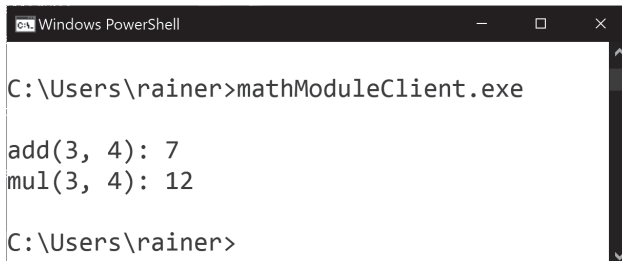
 std::cout << '\n';

 std::cout << "add(3, 4): " << add(3, 4) << '\n';
 std::cout << "mul(3, 4): " << mul(3, 4) << '\n';

}
```

---

Компиляция и выполнение этой программы дают ожидаемый результат.



```
Windows PowerShell

C:\Users\rainer>mathModuleClient.exe

add(3, 4): 7
mul(3, 4): 12

C:\Users\rainer>
```

Использование модулей и подмодулей



### Сборка модулей и подмодулей при помощи компилятора от Microsoft

Построение исполняемого файла из модулей и подмодулей

---

```
cl.exe /c /experimental:module mathModule1.ixx /EHsc
cl.exe /c /experimental:module mathModule2.ixx /EHsc
cl.exe /c /experimental:module mathModule.ixx /EHsc
cl.exe /EHsc /experimental:module mathModuleClient.cpp
 mathModule1.obj mathModule2.o\
bj mathModule.obj
```

---

При каждой компиляции создается два файла – IFC (файл интерфейса) \*.ifc, который неявно используется в последней строке сборки, и файл \*.obj, который явно используется в последней строке.

Я уже упоминал, что подмодуль – это тоже модуль. У каждого подмодуля есть раздел описания модуля. Соответственно, я могу создать вторую пользовательскую программу, которой нужен только модуль `math.math1`.

Главная программа использует лишь подмодуль `math.math1`

---

```
// mathModuleClient1.cpp

#include <iostream>

import math.math1;

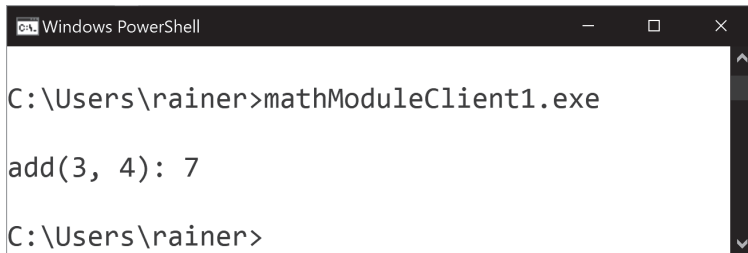
int main() {

 std::cout << '\n';

 std::cout << "add(3, 4): " << add(3, 4) << '\n';

}
```

---



```
Windows PowerShell

C:\Users\rainer>mathModuleClient1.exe

add(3, 4): 7

C:\Users\rainer>
```

Использование модулей и подмодулей

Разделение модулей на модули и подмодули – это способ разработчика модулей дать пользователю возможность использовать отдельные разделы модуля. Это правило не применимо к разделению модуля на разделы.

#### 4.2.8.2 Разделение модуля на разделы (modules partitions)

Модуль может быть разделен на разделы (partitions). Каждый раздел состоит из блока интерфейса модуля (файл интерфейса раздела) и ни одного или большего количества блоков реализации модуля. Те имена, которые разделы экспортируют, импортируются и реэкспортируются главным файлом интерфейса (primary interface file). Имена в разделах должны начинаться с имени модуля. Разделы модуля не могут существовать сами по себе.

Описание разделов модуля на самом деле сложнее для понимания, чем их реализация. Далее я перепишу модуль `math` и его подмодули `math.math1` и `math.math2` с использованием разделения модуля на разделы. В этом примере я буду использовать терминологию разделов модулей.

Главный файл интерфейса

---

```
1 // mathPartition.ixx
2
3 export module math;
4
5 export import :math1;
6 export import :math2;
```

---

Главный файл интерфейса модуля состоит из описания модуля (строка 3). Он импортирует и реэкспортирует разделы `math1` и `math2`, используя двоеточие (строки 5 и 6). Имена разделов должны начинаться с имени модуля. Соответственно, вам не нужно их указывать.

Первый раздел модуля

---

```
1 // mathPartition1.ixx
2
3 export module math:math1;
4
5 export int add(int fir, int sec) {
6 return fir + sec;
7 }
```

---

Второй раздел модуля

---

```
1 // mathPartition2.ixx
2
3 export module math:math2;
4
5 export {
6 int mul(int fir, int sec) {
7 return fir * sec;
8 }
9 }
```

---

Аналогично описанию модуля выражения `export module math:math1` и `export module math:math2` (строка 3) определяют интерфейс для разделов модуля. Интерфейс разделов модуля также является блоком интерфейса модуля. Имя `math` соответствует модулю, и имена `math1` и `math2` соответствуют разделам.

Импортируем разделы модуля

```
// mathModuleClient.cpp
```

```
import math;

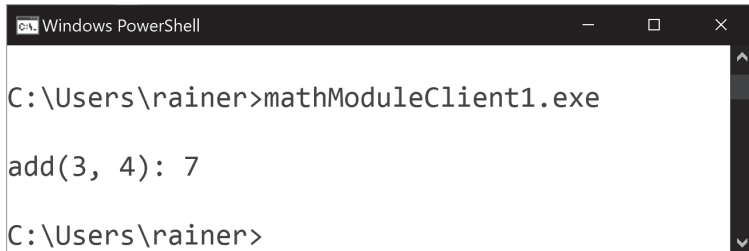
int main() {

 std::cout << '\n';

 std::cout << "add(3, 4): " << add(3, 4) << '\n';
 std::cout << "mul(3, 4): " << mul(3, 4) << '\n';

}
```

Вы могли уже заметить, что основная программа полностью идентична случаю программы, где было проиллюстрировано использование подмодулей. То же самое наблюдение справедливо и для создания исполняемого файла и его выполнения:



```
Windows PowerShell

C:\Users\rainer>mathModuleClient1.exe

add(3, 4): 7

C:\Users\rainer>
```

Использование модулей и подмодулей

## 4.2.9 Шаблоны в модулях

Я часто слышу следующий вопрос: как шаблоны экспортируются модулем? Когда вы инстанцируете шаблон, его определение должно быть доступно. Это та самая причина, по которой определения шаблонов находятся в заголовочных файлах. Использование шаблонов имеет следующую структуру.

### 4.2.9.0.1 Без модулей

- templateSum.h

Определение шаблонной функции `sum`

---

```
// templateSum.h
```

```
template <typename T, typename T2>
auto sum(T fir, T2 sec) {
 return fir + sec;
}
```

---

○ `sumMain.cpp`

Использование шаблонной функции `sum`

---

```
// sumMain.cpp
```

```
#include <templateSum.h>
```

```
int main() {

 sum(1, 1.5);

}
```

---

Программа `main` непосредственно включает заголовочный файл `templateSum.h`. Вызов `sum(1, 1.5)` порождает инстанцирование шаблона. В этом случае компилятор создает из шаблонной функции `sum` конкретную функцию `sum`, которая в качестве аргументов берет `int` и `double`. Если вы хотите посмотреть на этот процесс, то можете использовать данный пример на C++ Insights<sup>1</sup>.

#### 4.2.9.1 Использование шаблонных функций в модулях

В стандарте C++20 шаблоны могут и должны быть определены в модулях. У модулей есть уникальное внутреннее представление, которое не является ни исходным кодом, ни ассемблером. Это представление является чем-то вроде абстрактного синтаксического дерева (abstract syntax tree<sup>2</sup>, AST). Благодаря этому AST определения шаблонов доступны во время инстанцирования шаблонов.

В следующем примере я определяю шаблонную функцию `sum` в модуле `math`.

○ `mathModuleTemplate.ixx`

---

<sup>1</sup> <https://cppinsights.io/>.

<sup>2</sup> [https://en.wikipedia.org/wiki/Abstract\\_syntax\\_tree](https://en.wikipedia.org/wiki/Abstract_syntax_tree).

---

Определение шаблонной функции `sum`

---

```
// mathModuleTemplate.ixx

export module math;

export namespace math {

 template <typename T, typename T2>
 auto sum(T fir, T2 sec) {
 return fir + sec;
 }

}
```

---

○ clientTemplate.cpp

Использование шаблонной функции `sum`

---

```
// clientTemplate.cpp

#include <iostream>
import math;

int main() {

 std::cout << '\n';

 std::cout << "math::sum(2000, 11): " << math::sum(2000, 11) << '\n';

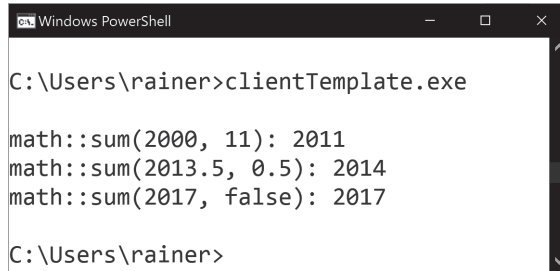
 std::cout << "math::sum(2013.5, 0.5): " << math::sum(2013.5, 0.5) << '\n';

 std::cout << "math::sum(2017, false): " << math::sum(2017, false) << '\n';

}
```

---

Команда для компиляции программы не отличается от той, что была ранее. Поэтому я пропущу ее и сразу же покажу результат работы программы:



```

C:\Users\rainer>clientTemplate.exe

math::sum(2000, 11): 2011
math::sum(2013.5, 0.5): 2014
math::sum(2017, false): 2017

C:\Users\rainer>

```

Использование шаблонной функции sum

С появлением модулей мы получили новый способ линковки программ.

### 4.2.10 Линковка на уровне модулей

До появления стандарта C++20 в C++ было два типа линковки – внутренняя и внешняя.

- **Внутренняя линковка** – имена с внутренней линковкой недоступны извне единицы компиляции. Внутренняя линковка включает в себя в первую очередь имена на уровне пространств имен, которые объявлены как статические (static), и членов анонимных пространств имен.
- **Внешняя линковка** – имена с внешней линковкой доступны извне единицы компиляции. Внешняя линковка включает имена, которые не были объявлены как статические, классы и их члены, а также переменные и шаблоны.

Модули добавляют новый тип линковки – линковку модулей (module linkage):

- **линковка модулей** – имена с линковкой модулей доступны только внутри модуля. У имен задана линковка модуля, если они не имеют внешней линковки и не экспортируются.

Я проиллюстрирую это небольшим изменением предыдущего модуля – mathModuleTemplate.ixx. Представьте, что я хочу вернуть пользователю моей шаблонной функции sum не только результат сложения, но и определяемый компилятором тип результата.

Улучшенное определение шаблонной функции sum

---

```

1 // mathModuleTemplate1.ixx
2
3 module;
4
5 #include <iostream>
6 #include <typeinfo>
7 #include <utility>
8
9 export module math;
10

```



---

```

11 template <typename T>
12 auto showType(T&& t) {
13 return typeid(std::forward<T>(t)).name();
14 }
15
16 export namespace math {
17
18 template <typename T, typename T2>
19 auto sum(T fir, T2 sec) {
20 auto res = fir + sec;
21 return std::make_pair(res, showType(res));
22 }
23
24 }

```

---

Вместо суммы двух чисел шаблонная функция `sum` теперь возвращает `std::pair`<sup>1</sup> (строка 21), состоящую из суммы и строчного представления типа значения `res`. Обратите внимание, что я поместил шаблонную функцию `showType` (строка 11) вне экспортируемого пространства имен `math` (строка 16). Поэтому вызов его извне модуля `math` невозможен. Шаблонная функция `showType` использует идеальную пересылку (perfect forwarding)<sup>2</sup> для сохранения типа значения аргумента `t`. Оператор `typeid`<sup>3</sup> запрашивает информацию о типе во время выполнения (RTTI<sup>4</sup>).

Использование улучшенной шаблонной функции `sum`

---

```

1 // clientTemplate1.cpp
2
3 #include <iostream>
4 import math;
5
6 int main() {
7
8 std::cout << '\n';
9
10 auto [val, message] = math::sum(2000, 11);

```

---

<sup>1</sup> <https://en.cppreference.com/w/cpp/utility/pair>.

<sup>2</sup> <https://www.modernescpp.com/index.php/perfect-forwarding>.

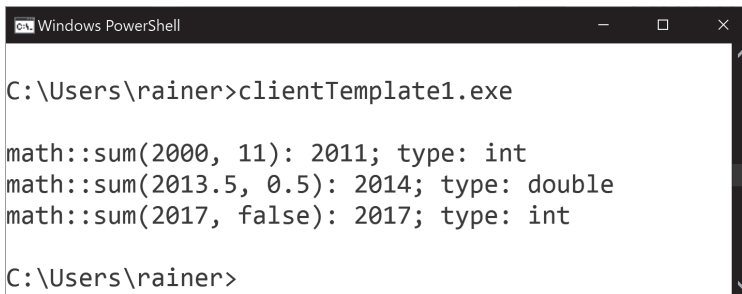
<sup>3</sup> <https://en.cppreference.com/w/cpp/language/typeid>.

<sup>4</sup> <https://en.cppreference.com/w/cpp/types>.

```
11 std::cout << "math::sum(2000, 11): " << val << " ";
 type: " << message << '\n';
12
13 auto [val1, message1] = math::sum(2013.5, 0.5);
14 std::cout << "math::sum(2013.5, 0.5): " << val1
15 << " "; type: " << message1
16 << '\n';
17 auto [val2, message2] = math::sum(2017, false);
18 std::cout << "math::sum(2017, false): " << val2 << " "; type:
19 " << message2
20 << '\n';
21 }
```

---

Теперь программа возвращает как значение суммы, так и строковое представление автоматически определенного типа значения.



```
Windows PowerShell
C:\Users\rainer>clientTemplate1.exe

math::sum(2000, 11): 2011; type: int
math::sum(2013.5, 0.5): 2014; type: double
math::sum(2017, false): 2017; type: int

C:\Users\rainer>
```

Использование улучшенной шаблонной функции sum

### 4.2.11 Заголовочные блоки

К моменту подготовки этой книги (конец 2020 года) ни один компилятор не поддерживал заголовочные блоки (header units). Заголовочные блоки – это такой способ плавного перехода от заголовочных файлов к модулям. Вам нужно будет просто заменить директиву `#include` новой директивой `import`.

Замена директивы `#include` директивой `import`

---

```
#include <vector> => import <vector>;
#include "myHeader.h" => import "myHeader.h";
```

---

Что дает такая замена? `Import` использует те же правила поиска, что и `#include`. Это значит, что в случае кавычек ("myHeader.h") поиск сначала происходит в локальном каталоге, прежде чем он пойдет по системному пути поиска.

Кроме того, `import` – это больше, чем просто замена текста. В этом случае компилятор генерирует что-то, похожее на модуль из директивы `import`, и рассматривает результат как модуль. Оператор импорта получает все экспорти-

руемые имена из заголовочного файла. Эти экспортируемые имена включают в себя макросы. Импорт подобных заголовочных блоков будет быстрее, чем традиционное подключение заголовочных файлов, и сравнимо по скорости с предварительно скомпилированными и заголовочными файлами.

#### 4.2.11.1 Единственный недостаток

В использовании заголовочных блоков есть один недостаток. Не все заголовочные файлы можно импортировать. Какие именно заголовочные файлы можно импортировать, зависит от реализации<sup>1</sup>, но стандарт C++ гарантирует, что все заголовочные файлы стандартной библиотеки являются импортируемыми. Возможность импортировать не относится к заголовочным файлам C. Они просто заворачиваются в пространство имен `std`. Например, `<cstring>` – это C++-вариант `<string.h>`. Вы можете легко опознать такие файлы, поскольку теперь они из вида `xxx.h` стали выглядеть как `sxxx`.

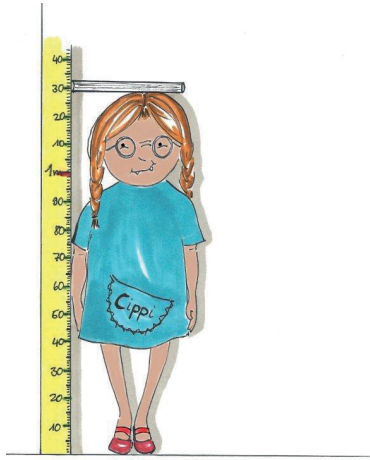


#### Важные замечания

- ♦ Модули исправляют недостатки заголовочных файлов и макросов. Их импорт практически ничего не стоит, и, в отличие от макросов, последовательность, в которой мы подключаем заголовочные файлы, не влияет на результат. Кроме того, не возникают коллизии имен.
- ♦ Модуль состоит из блока интерфейса модуля и блока реализации модуля. Должен быть как минимум один блок интерфейса, содержащий описание экспорта модуля и произвольное количество блоков реализации модуля. Неэкспортируемые имена имеют линковку модуля и не могут быть использованы вне модуля.
- ♦ В модулях могут быть заголовочные файлы, и они могут импортировать и реэкспортировать другие модули.
- ♦ Стандартная библиотека в C++20 не реализована в виде модулей. Построение собственных модулей может быть сложной задачей.
- ♦ Для структурирования больших программных систем модули предоставляют две возможности: использование подмодулей и разделов модуля. В отличие от разделов модуля, подмодули сами являются модулями.
- ♦ Благодаря заголовочным блокам вы можете заменить оператор `include` на оператор `import`, и компилятор сам создаст модуль.

<sup>1</sup> <https://en.cppreference.com/w/cpp/language/ub>.

## 4.3 Оператор трехстороннего сравнения



Сиппи измеряет свой рост

Оператор трехстороннего сравнения за его внешний вид `<=>` часто называют космическим кораблем (spaceship operator). Этот оператор для двух значений  $A$  и  $B$  определяет, какое из следующих соотношений —  $A < B$ ,  $A == B$  или  $A > B$  — имеет место. Вы можете сами определить этот оператор, либо компилятор сгенерирует его за вас.

Для того чтобы оценить все преимущества, даваемые оператором трехстороннего сравнения, давайте начнем с классического способа реализации.

### 4.3.1 Упорядочение до C++20

Я реализовал простую обертку на `int` в виде структуры `MyInt`. И конечно, я хочу сравнивать между собой экземпляры этого типа. Ниже приводится мой вариант сравнения при помощи шаблонной функции `isLessThan`.

`MyInt` поддерживает сравнение «меньше чем»

---

```
// comparisonOperator.cpp
```

```
#include <iostream>
```

```
struct MyInt {
 int value;
 explicit MyInt(int val): value{val} { }
 bool operator < (const MyInt& rhs) const {
 return value < rhs.value;
 }
};
```

```

template <typename T>
constexpr bool isLessThan(const T& lhs, const T& rhs) {
 return lhs < rhs;
}

int main() {

 std::cout << std::boolalpha << '\n';

 MyInt myInt2011(2011);
 MyInt myInt2014(2014);

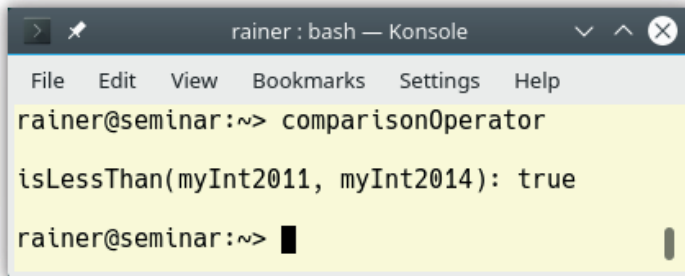
 std::cout << "isLessThan(myInt2011, myInt2014): "
 << isLessThan(myInt2011, myInt2014) << '\n';

 std::cout << '\n';

}

```

Программа работает, как и ожидается:



```

rainer : bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~> comparisonOperator
isLessThan(myInt2011, myInt2014): true
rainer@seminar:~> █

```

Использование оператора «меньше чем»

Если честно, то `MyInt` — абсолютно не интуитивный тип. Когда вы определяете одно из шести отношений сравнения, вам нужно определить их все. Интуитивные типы должны быть как минимум частично регулярными (*semiregular*). Теперь мне придется написать много кода. Вот пропущенные пять операторов.

Пропущенные пять операторов сравнения

---

```
bool operator == (const MyInt& rhs) const {
 return value == rhs.value;
}
bool operator != (const MyInt& rhs) const {
 return !(*this == rhs);
}
bool operator <= (const MyInt& rhs) const {
 return !(rhs < *this);
}
bool operator > (const MyInt& rhs) const {
 return rhs < *this;
}
bool operator >= (const MyInt& rhs) const {
 return !(*this < rhs);
}
```

---

Теперь давайте перейдем к стандарту C++20 и посмотрим на оператор трехстороннего сравнения.

### 4.3.2 Упорядочение начиная со стандарта C++20

Вы можете определить оператор трехстороннего сравнения или потребовать, чтобы компилятор его сгенерировал сам при помощи `= default`. В обоих случаях вы автоматически получаете все шесть операций сравнения: `==`, `!=`, `<`, `<=`, `>=` и `>`.

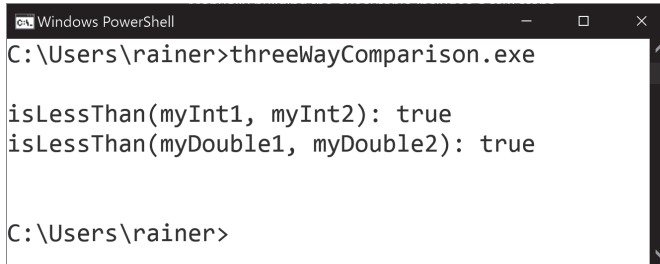
Реализация или запрос трехстороннего сравнения

---

```
1 // threeWayComparison.cpp
2
3 #include <compare>
4 #include <iostream>
5
6 struct MyInt {
7 int value;
8 explicit MyInt(int val): value{val} { }
9 auto operator<=>(const MyInt& rhs) const {
10 return value <=> rhs.value;
11 }
12 };
13
```

```
14 struct MyDouble {
15 double value;
16 explicit constexpr MyDouble(double val): value{val} { }
17 auto operator<=>(const MyDouble&) const = default;
18 };
19
20 template <typename T>
21 constexpr bool isLessThan(const T& lhs, const T& rhs) {
22 return lhs < rhs;
23 }
24
25 int main() {
26
27 std::cout << std::boolalpha << '\n';
28
29 MyInt myInt1(2011);
30 MyInt myInt2(2014);
31
32 std::cout << "isLessThan(myInt1, myInt2): "
33 << isLessThan(myInt1, myInt2) << '\n';
34
35 MyDouble myDouble1(2011);
36 MyDouble myDouble2(2014);
37
38 std::cout << "isLessThan(myDouble1, myDouble2): "
39 << isLessThan(myDouble1, myDouble2) << '\n';
40
41 std::cout << '\n';
42
43 }
```

Определенный пользователем (строка 9) и созданный компилятором (строка 17) операторы трехстороннего сравнения работают, как и ожидалось.



```

C:\Users\rainer>threeWayComparison.exe

isLessThan(myInt1, myInt2): true
isLessThan(myDouble1, myDouble2): true

C:\Users\rainer>

```

Использование заданного пользователем и сгенерированного компилятором оператора трехстороннего сравнения

В этом случае есть тонкое различие между заданным пользователем и сгенерированным компилятором операторами. Выводимый компилятором тип для `MyInt` (строка 9) поддерживает строгое упорядочение, а созданный компилятором оператор для класса `MyDouble` (строка 17) поддерживает лишь частичное упорядочение.



### Автоматическое сравнение указателей

Сгенерированный компилятором оператор трехстороннего сравнения сравнивает указатели, а не объекты, на которые они ссылаются.

Автоматическое сравнение указателей

---

```

1 // spaceshipPoiner.cpp
2
3 #include <iostream>
4 #include <compare>
5 #include <vector>
6
7 struct A {
8 std::vector<int>* pointerToVector;
9 auto operator <=> (const A&) const = default;
10 };
11
12 int main() {
13
14 std::cout << '\n';
15
16 std::cout << std::boolalpha;
17
18 A a1{new std::vector<int>()};
19 A a2{new std::vector<int>()};
20
21 std::cout << "(a1 == a2): " << (a1 == a2) << "\n\n";
22
23 }

```

---



Как это ни странно, результат сравнения `a1 == a2` (строка 21) равен `false`, а не `true`, поскольку сравниваются адреса `std::vector<int>*`.

```
(a1 == a2): false
```

Сравнение указателей

Есть три категории сравнения.

### 4.3.3 Категории сравнения

Три категории сравнения называются строгим (strong ordering), слабым (weak ordering) и частичным упорядочением (partial ordering). Для типа `T` следующие три свойства разделяют три категории сравнения.

1. `T` поддерживает все шесть операторов сравнения: `==`, `!=`, `<`, `<=`, `>` и `>=` (операторы сравнения).
2. Все эквивалентные значения неразличимы (эквивалентность).
3. Все значения из `T` сравнимы между собой: для произвольных значений `a` и `b` типа `T` только одно из следующих трех отношений справедливо: `a < b`, `a == b` и `a > b` (сравнимость).

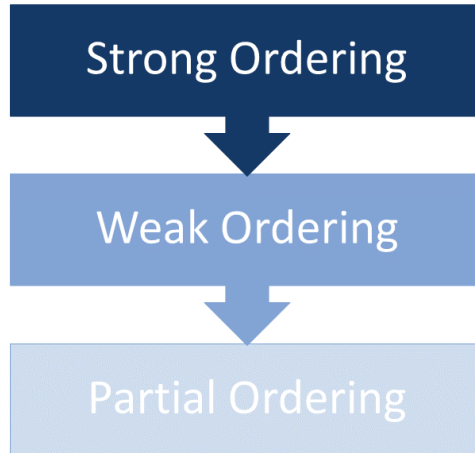
Когда вы используете в качестве критерия сортировки нечувствительные к регистру представления строк, эквивалентные значения не обязательно должны быть одинаковыми. Точно так же два значения с плавающей точкой не обязательно должны быть сравнимыми: для `a = 5.5` и `b = NaN` (Not a Number) ни одно из следующих выражений не возвращает `true`: `a < NaN`, `a == NaN` и `a > NaN`.

Исходя из этих трех свойств, стратегии сравнения легко различаются.

Строгое, слабое и частичное упорядочение

| Категория сравнения    | Операторы сравнения | Эквивалентность | Сравнимость |
|------------------------|---------------------|-----------------|-------------|
| Строгое упорядочение   | Да                  | Да              | Да          |
| Слабое упорядочение    | Да                  |                 | Да          |
| Частичное упорядочение | Да                  |                 |             |

Тип, поддерживающий строгое упорядочение, также поддерживает и слабое, и частичное. Тип, поддерживающий слабое упорядочение, поддерживает частичное. В обратную сторону это не работает.



Строгое, слабое и частичное упорядочение

Если объявленный возвращаемый тип – это `auto`, то реальный возвращаемый тип определяется из общей стратегии сравнения подобъектов и элементов массива, которые сравниваются между собой.

Давайте рассмотрим пример к этому правилу.

Реализуем или вызовем трехсторонний оператор сравнения

---

```
1 // strongWeakPartial.cpp
2
3 #include <compare>
4
5 struct Strong {
6 std::strong_ordering operator <=> (const Strong&) const = d
7 };
8
9 struct Weak {
10 std::weak_ordering operator <=> (const Weak&) const = defau
11 };
12
13 struct Partial {
14 std::partial_ordering operator <=> (const Partial&) const =
15 };
16
17 struct StrongWeakPartial {
18
19 Strong s;
```

---

```

20 Weak w;
21 Partial p;
22
23 auto operator <=> (const StrongWeakPartial&) const = default;
24
25 // FINE
26 // std::partial_ordering operator <=> (const StrongWeakParti
27 ;
28
29 // ERROR
30 // std::strong_ordering operator <=> (const StrongWeakPartia
31
32 // std::weak_ordering operator <=> (const StrongWeakPartial&
33
34
35 };
36
37 int main() {
38
39 StrongWeakPartial a1, a2;
40
41 a1 < a2;
42
43 }
```

---

Тип `StrongWeakPartial` содержит подобъекты, поддерживающие строгое (строка 6), слабое (строка 10) и частичное (строка 14) упорядочение. Общей категорией сравнения для типа `StrongWeakPartial` (строка 17) будет `std::partial_ordering`. Использование другой категории сравнения, такой как строгое упорядочение (строка 29) или слабое упорядочение (строка 30), приведет к ошибке компиляции.

Теперь я хочу рассмотреть создаваемый компилятором оператор трехстороннего сравнения.

### 4.3.4 Создаваемый компилятором оператор трехстороннего сравнения

Создаваемый компилятором оператор трехстороннего сравнения требует заголовочного файла `<compare>`, неявно является `constexpr` и `noexcept`<sup>1</sup> и выполняет лексикографическое сравнение.

---

<sup>1</sup> <https://www.modernescpp.com/index.php/c-core-guidelines-the-noexcept-specifier-and-operator>.

Вы даже непосредственно можете использовать оператор трехстороннего сравнения.

#### 4.3.4.1 Непосредственное использование оператора трехстороннего сравнения

Программа `spaceship.cpp` непосредственно использует оператор трехстороннего сравнения.

Реализация или вызов оператора трехстороннего сравнения

---

```
1 // spaceship.cpp
2
3 #include <compare>
4 #include <iostream>
5 #include <string>
6 #include <vector>
7
8 int main() {
9
10 std::cout << '\n';
11
12 int a(2011);
13 int b(2014);
14 auto res = a <=> b;
15 if (res < 0) std::cout << "a < b" << '\n';
16 else if (res == 0) std::cout << "a == b" << '\n';
17 else if (res > 0) std::cout << "a > b" << '\n';
18
19 std::string str1("2014");
20 std::string str2("2011");
21 auto res2 = str1 <=> str2;
22 if (res2 < 0) std::cout << "str1 < str2" << '\n';
23 else if (res2 == 0) std::cout << "str1 == str2" << '\n';
24 else if (res2 > 0) std::cout << "str1 > str2" << '\n';
25
26 std::vector<int> vec1{1, 2, 3};
27 std::vector<int> vec2{1, 2, 3};
28 auto res3 = vec1 <=> vec2;
29 if (res3 < 0) std::cout << "vec1 < vec2" << '\n';
30 else if (res3 == 0) std::cout << "vec1 == vec2" << '\n';
```

---

```

31 else if (res3 > 0) std::cout << "vec1 > vec2" << '\n';
32
33 std::cout << '\n';
34
35 }
```

---

Программа использует оператор трехстороннего сравнения для `int` (строка 14), `string` (строка 21) и `vector` (строка 28). Ниже приводится вывод этой программы.

```

a < b
str1 > str2
vec1 == vec2
```

Непосредственное использование оператора трехстороннего сравнения

Как уже было упомянуто, все эти сравнения `constexpr` и могут быть выполнены во время компиляции.

#### 4.3.4.2 Сравнение во время компиляции

Оператор трехстороннего сравнения неявно является `constexpr`. Поэтому я могу упростить предыдущую программу `threeWayComparison.cpp` и сравнить `MyDouble` во время компиляции.

Создаваемый компилятором `constexpr` трехсторонний оператор сравнения

---

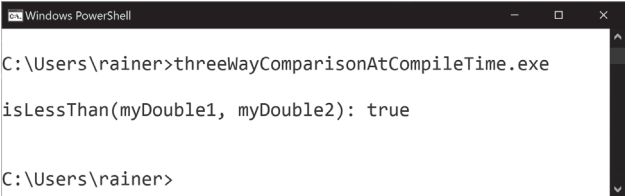
```

1 // threeWayComparisonAtCompileTime.cpp
2
3 #include <compare>
4 #include <iostream>
5
6 struct MyDouble {
7 double value;
8 explicit constexpr MyDouble(double val): value{val} { }
9 auto operator<=>(const MyDouble&) const = default;
10 };
11
12 template <typename T>
13 constexpr bool isLessThan(const T& lhs, const T& rhs) {
14 return lhs < rhs;
15 }
16
```

```
17 int main() {
18
19 std::cout << std::boolalpha << '\n';
20
21 constexpr MyDouble myDouble1(2011);
22 constexpr MyDouble myDouble2(2014);
23
24 constexpr bool res = isLessThan(myDouble1, myDouble2);
25
26 std::cout << "isLessThan(myDouble1, myDouble2): "
27 << res << '\n';
28
29 std::cout << '\n';
30
31 }
```

---

Я запрашиваю результат сравнения во время компиляции, и я его получаю.



```
Windows PowerShell
C:\Users\rainer>threeWayComparisonAtCompileTime.exe

isLessThan(myDouble1, myDouble2): true

C:\Users\rainer>
```

Использование сгенерированного компилятором constexpr оператора трехстороннего сравнения

#### 4.3.4.3 Лексикографическое сравнение

Создаваемый компилятором оператор трехстороннего сравнения выполняет лексикографическое сравнение. Лексикографическое в этом случае значит, что все базовые классы сравниваются слева направо и все нестатические члены класса сравниваются в порядке их объявления. Я должен уточнить: для быстрого действия генерируемые компилятором операторы == и != в C++20 ведут себя иначе. Я напишу об этом в разделе «Оптимизированные операторы == и !=».

Публикация «Simplify Your Code With Rocket Science: C++ 20's Spaceship operator»<sup>1</sup> в блоге Microsoft Team дает хороший пример лексикографического сравнения. Для улучшения читаемости я добавил несколько комментариев.

---

<sup>1</sup> <https://devblogs.microsoft.com/cppblog/simplify-your-code-with-rocket-science-c20s-spaceship-operator/>.

## Лексикографическое сравнение

---

```

1 struct Basics {
2 int i;
3 char c;
4 float f;
5 double d;
6 auto operator<=>(const Basics&) const = default;
7 };
8
9 struct Arrays {
10 int ai[1];
11 char ac[2];
12 float af[3];
13 double ad[2][2];
14 auto operator<=>(const Arrays&) const = default;
15 };
16
17 struct Bases : Basics, Arrays {
18 auto operator<=>(const Bases&) const = default;
19 };
20
21 int main() {
22 constexpr Basics a = { { 0, 'c', 1.f, 1. }, //Basics
23 { { 1 }, { 'a', 'b' }, { 1.f, 2.f, 3.f }, //Arrays
24 { { 1., 2. }, { 3., 4. } } } };
25 constexpr Basics b = { { 0, 'c', 1.f, 1. }, //Basics
26 { { 1 }, { 'a', 'b' }, { 1.f, 2.f, 3.f }, //Arrays
27 { { 1., 2. }, { 3., 4. } } } };
28 static_assert(a == b);
29 static_assert(!(a != b));
30 static_assert(!(a < b));
31 static_assert(a <= b);
32 static_assert(!(a > b));
33 static_assert(a >= b);
34 }

```

Я полагаю, что наиболее сложным аспектом этой программы является не оператор трехстороннего сравнения, а инициализация `Bases` через агрегированную инициализацию (строки 22 и 25). Агрегированная инициализация позволяет нам непосредственно проинициализировать все члены класса (`class`, `struct`, `union`), когда они являются `public`. В этом случае вы можете использовать инициализацию при помощи фигурных скобок. Агрегированная инициализация более подробно будет рассматриваться в разделе про назначенную инициализацию (`designated initialization`).



### Оптимизированные операторы == и !=

Существует возможность оптимизации для типов вроде строк и векторов. В этом случае операторы == и != могут быть быстрее, чем созданный компилятором оператор трехстороннего сравнения. Операторы == и != могут прекратить работу, если значения имеют разные длины. В противном случае, если одно значение является префиксом для другого, то лексикографическое сравнение сравнит все элементы до окончания более короткого значения. Комитет по стандартизации знает об этом и исправил эту особенность реализации оператора в документе P1185R2<sup>1</sup>. Соответственно, сгенерированные компилятором операторы == и != в случае строчных или векторизированных типов (похожих на vector) сперва сравнивают длины и только потом при необходимости – значения.

Теперь пришло время для чего-то нового в стандарте C++. В C++20 впервые вводится понятие переписывания выражений (rewriting expressions).

## 4.3.5 Переписывание выражений

Когда компилятор видит что-то вроде  $a < b$ , то он переписывает (rewrite) это при помощи оператора трехстороннего сравнения в виде  $(a <=> b) < 0$ .

Конечно, это правило применимо ко всем шести операторам сравнения.

Выражение  $a \text{ OP } b$  становится  $(a <=> b) \text{ OP } 0$ , если нет преобразований типа от  $\text{type}(a)$  к  $\text{type}(b)$ , то компилятор генерирует новое выражение  $0 \text{ OP } (b <=> a)$ .

Например, это значит, что для оператора «меньше чем», если  $(a <=> b) < 0$  не работает, компилятор генерирует  $0 < (b <=> a)$ . Таким образом компилятор заботится о симметрии операторов сравнения.

Вот несколько примеров переписывания выражений.

Переписывание выражений для MyInt

```
1 // rewritingExpressions.cpp
2
3 #include <compare>
4 #include <iostream>
5
6 class MyInt {
7 public:
8 constexpr MyInt(int val): value{val} { }
9 auto operator<=>(const MyInt& rhs) const = default;
10 private:
11 int value;
12 };
13
14 int main() {
15
```

<sup>1</sup> <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1185r2.html>.



```

16 std::cout << '\n';
17
18 constexpr MyInt myInt2011(2011);
19 constexpr MyInt myInt2014(2014);
20
21 constexpr int int2011(2011);
22 constexpr int int2014(2014);
23
24 if (myInt2011 < myInt2014) std::cout << "myInt2011 < myInt2014" << '\n';
25 if ((myInt2011 <=> myInt2014) < 0) std::cout << "myInt2011 < myInt2014" << '\n';
26
27 std::cout << '\n';
28
29 if (myInt2011 < int2014) std::cout << "myInt2011 < int2014" << '\n';
30 if ((myInt2011 <=> int2014) < 0) std::cout << "myInt2011 < int2014" << '\n';
31
32 std::cout << '\n';
33
34 if (int2011 < myInt2014) std::cout << "int2011 < myInt2014" << '\n';
35 if (0 < (myInt2014 <=> int2011)) std::cout << "int2011 < myInt2014" << '\n';
36
37 std::cout << '\n';
38
39 }

```

Я использовал в строках 24, 29 и 34 оператор «меньше чем» и соответствующее выражение через оператор трехстороннего сравнения. Наиболее интересной является строка 35. Она показывает, как сравнение (`int2011 < myInt2014`) приводит к генерации выражения (`0 < (myInt2014 <=> int2011)`) через оператор трехстороннего сравнения.

```

myInt2011 < myInt2014
myInt2011 < myInt2014

myInt2011 < int2014
myInt2011 < int2014

int2011 < myInt2014
int2011 < myInt2014

```

Переписывание выражений

Если честно, то у `MyInt` есть одна проблема: его конструктор, принимающий один аргумент, должен быть объявлен как `explicit`. Конструкторы, принимающие всего один аргумент, такие как `MyInt (int val)`, являются конструкторами преоб-

разования. Это значит, что экземпляр `MyInt` может быть получен из любого целочисленного или вещественного значения, поскольку каждое целочисленное значение или значение с плавающей точкой может быть неявно переведено в `int`.

Давайте это исправим и сделаем конструктор `MyInt (int val) explicit`. Для поддержки сравнения `MyInt` и `int` нам понадобится дополнительный оператор трехстороннего сравнения с переменной типа `int`.

Дополнительный оператор трехстороннего сравнения с `int`

```
myInt2011 < myInt2014
```

```
myInt2011 < myInt2014
```

```
myInt2011 < int2014
```

```
myInt2011 < int2014
```

```
int2011 < myInt2014
```

```
22 }
23
24 int main() {
25
26 std::cout << std::boolalpha << '\n';
27
28 constexpr MyInt myInt2011(2011);
29 constexpr MyInt myInt2014(2014);
30
31 constexpr int int2011(2011);
32 constexpr int int2014(2014);
33
```

```

34 std::cout << "isLessThan(myInt2011, myInt2014): "
35 << isLessThan(myInt2011, myInt2014) << '\n';
36
37 std::cout << "isLessThan(int2011, myInt2014): "
38 << isLessThan(int2011, myInt2014) << '\n';
39
40 std::cout << "isLessThan(myInt2011, int2014): "
41 << isLessThan(myInt2011, int2014) << '\n';
42
43 constexpr auto res = isLessThan(myInt2011, int2014);
44
45 std::cout << '\n';
46
47 }

```

Я определил (строка 10) оператор трехстороннего сравнения и объявил его как `constexpr`. Задаваемый пользователем оператор трехстороннего сравнения не является по умолчанию `constexpr`, в отличие от генерируемого компилятором. Сравнение `MyInt` и `int` возможно в каждой комбинации (строки 34, 37 и 40).

```

22 }
23
24 int main() {
25
26 std::cout << std::boolalpha << '\n';
27
28 constexpr MyInt myInt2011(2011);
29 constexpr MyInt myInt2014(2014);
30
31 // ...

```

Оператор трехстороннего сравнения для `int`

Реализация различных операторов трехстороннего сравнения выглядит очень элегантно. Компилятор сам создает сравнения для `MyInt`, и пользователь явно определяет сравнение с `int`. Кроме того, благодаря переупорядочиванию нам нужно определить только два оператора, а не все  $18 = 3 * 6$  комбинаций. Здесь 3 обозначает комбинации `int OP MyInt`, `MyInt OP MyInt` и `MyInt OP int` и 6 – это количество операторов сравнения.

### 4.3.6 Задаваемые пользователем и создаваемые автоматически операторы сравнения

Когда вы можете определить один из шести операторов сравнения и также автоматически сгенерировать их все с помощью оператора трехстороннего сравнения, возникает один вопрос: у кого из них приоритет выше? Например, у рассматриваемой реализации `MyInt` есть задаваемый пользователем оператор «меньше или равно», а также шесть операторов сравнения, созданных компилятором.

Взаимодействие задаваемых пользователем и автоматически создаваемых операторов

---

```
1 // userDefinedAutoGeneratedOperators.cpp
2
3 #include <compare>
4 #include <iostream>
5
6 class MyInt {
7 public:
8 constexpr explicit MyInt(int val): value{val} { }
9 bool operator == (const MyInt& rhs) const {
10 std::cout << "==" << '\n';
11 return value == rhs.value;
12 }
13 bool operator < (const MyInt& rhs) const {
14 std::cout << "<" << '\n';
15 return value < rhs.value;
16 }
17
18 auto operator<=>(const MyInt& rhs) const = default;
19
20 private:
21 int value;
22 };
23
24 int main() {
25
26 MyInt myInt2011(2011);
27 MyInt myInt2014(2014);
28
29 myInt2011 == myInt2014;
30 myInt2011 != myInt2014;
31 myInt2011 < myInt2014;
32 myInt2011 <= myInt2014;
33 myInt2011 > myInt2014;
34 myInt2011 >= myInt2014;
35
36 }
```

---

Для того чтобы увидеть задаваемые пользователем операторы `==` и `<` в действии, я добавил соответствующее сообщение в `std::cout`. Ни один из этих операторов не может быть `constexpr`, так как `std::cout` – это операция времени выполнения.

Давайте посмотрим, что происходит при вызове всех операторов.

```
==
==
<
```

Задаваемый пользователем и автоматически создаваемые операторы

В этом случае компилятор использует задаваемые пользователем оператор `==` (строки 29 и 30) и оператор `<` (строка 31). Кроме того, компилятор синтезирует оператор `!=` (строка 30) из оператора `==`. С другой стороны, компилятор не создает оператор `==` из оператора `!=`.



#### Схожесть с Python

В Python 3 компилятор генерирует при необходимости `!=` из `==`, но не наоборот. В Python 2 сравнение (так называемое «богатое сравнение», *rich comparison*) при помощи задаваемых пользователем шести операторов сравнения имеет более высокий приоритет, чем оператор трехстороннего сравнения `__cmp__`. Упомянется Python 2, потому что этот оператор был удален в Python 3.



#### Важные замечания

- ♦ По умолчанию принято, что для оператора `<=>` компилятор генерирует все шесть операторов сравнения. Сгенерированные компилятором операторы сравнения используют лексикографическое сравнение: все базовые классы сравниваются слева направо и все нестатические члены класса – в порядке их объявления.
- ♦ Когда присутствуют и сгенерированные компилятором, и заданные пользователем операторы сравнения, то заданные пользователем операторы сравнения имеют более высокий приоритет.
- ♦ Компилятор перестраивает выражения для учета симметричности операторов сравнения. Например, если `(a<=>b) < 0` не работает, то компилятор использует `0<(b<=>a)`.

## 4.4 Назначенная инициализация



Сиппи получает божественное прикосновение

Назначенная инициализация (designated initialization) – это случай агрегированной инициализации. Поэтому писать что-то о назначенной инициализации означает писать об агрегированной инициализации.

### 4.4.1 Агрегированная инициализация

Во-первых, что значит агрегат? Агрегаты – это массивы и классы. А классы, в свою очередь, могут иметь тип класса, структуры или объединения (union).

В стандарте C++20 для классов, поддерживающих агрегированную инициализацию, должны быть выполнены следующие условия:

- нет приватных (private) или защищенных (protected) нестатических членов данных;
- нет задаваемых пользователем или унаследованных конструкторов (inherited constructors);
- нет виртуальных (virtual), приватных или защищенных базовых классов;
- нет виртуальных функций.

Следующая программа демонстрирует пример агрегированной инициализации.

Агрегированная инициализация

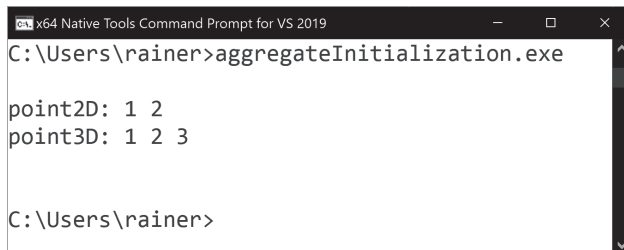
```
1 // aggregateInitialization.cpp
2
3 #include <iostream>
4
5 struct Point2D{
6 int x;
7 int y;
8 };
9
```

```

10 class Point3D{
11 public:
12 int x;
13 int y;
14 int z;
15 };
16
17 int main(){
18
19 std::cout << '\n';
20
21 Point2D point2D{1, 2};
22 Point3D point3D{1, 2, 3};
23
24 std::cout << "point2D: " << point2D.x << " " << point2D.y << '\n';
25 std::cout << "point3D: " << point3D.x << " " << point3D.y << " "
26 << point3D.z << '\n';
27
28 std::cout << '\n';
29
30 }

```

Строки 21 и 22 непосредственно инициализируют агрегатные объекты при помощи фигурных скобок. Последовательность инициализирующих значений в фигурных скобках должна точно соответствовать порядку следования членов. В разделе, посвященном трехстороннему оператору сравнения, мы рассматривали более сложный пример агрегатной инициализации.



```

C:\Users\rainer>aggregateInitialization.exe

point2D: 1 2
point3D: 1 2 3

C:\Users\rainer>

```

Агрегатная инициализация

На основе агрегатной инициализации из стандарта C++11 в стандарт C++20 была добавлена назначенная инициализация. В конце 2020 года только компилятор от Microsoft полностью поддерживал такую инициализацию.

### 4.4.2 Именованная инициализация членов класса

Назначенная инициализация позволяет напрямую инициализировать члены типа класса, используя их имена. Для объединений может быть задан только один инициализатор. Как и в агрегатной инициализации, последовательность инициализирующих значений в фигурных скобках должна совпадать с порядком объявления членов класса.

Назначенная инициализация

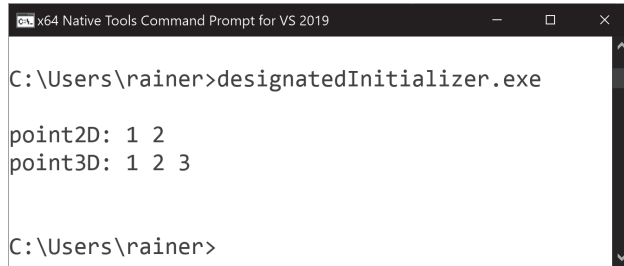
---

```
1 // designatedInitializer.cpp
2
3 #include <iostream>
4
5 struct Point2D{
6 int x;
7 int y;
8 };
9
10 class Point3D{
11 public:
12 int x;
13 int y;
14 int z;
15 };
16
17 int main(){
18
19 std::cout << '\n';
20
21 Point2D point2D{.x = 1, .y = 2};
22 Point3D point3D{.x = 1, .y = 2, .z = 3};
23
24 std::cout << "point2D: " << point2D.x << " " << point2D.y << '\n';
25 std::cout << "point3D: " << point3D.x << " " << point3D.y << " "
26 << point3D.z << '\n';
27
28 std::cout << '\n';
29
30 }
```

---

Строки 21 и 22 используют назначенные инициализаторы для инициализации агрегатных объектов. Такие инициализаторы, как `.x` и `.y`, часто называются указателями на объект инициализации (designators).





```

C:\Users\rainer>designatedInitializer.exe

point2D: 1 2
point3D: 1 2 3

C:\Users\rainer>

```

#### Именованные инициализаторы

Члены агрегатного объекта могут иметь значения по умолчанию. Эти значения по умолчанию используются, когда соответствующий инициализатор пропущен. Но это не работает для объединений.

Именованные указатели со значениями по умолчанию

---

```

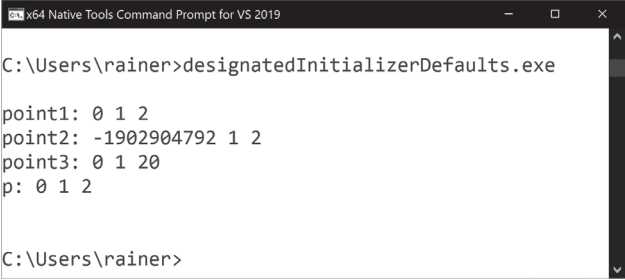
1 // designatedInitializersDefaults.cpp
2
3 #include <iostream>
4
5 class Point3D{
6 public:
7 int x;
8 int y = 1;
9 int z = 2;
10 };
11
12 void needPoint(Point3D p) {
13 std::cout << "p: " << p.x << " " << p.y << " " << p.z << '\n';
14 }
15
16 int main(){
17
18 std::cout << '\n';
19
20 Point3D point1{.x = 0, .y = 1, .z = 2};
21 std::cout << "point1: " << point1.x << " " << point1.y << " "
22 << point1.z << '\n';
23
24 Point3D point2;
25 std::cout << "point2: " << point2.x << " " << point2.y << " "
26 << point2.z << '\n';

```

```
27
28 Point3D point3{.x = 0, .z = 20};
29 std::cout << "point3: " << point3.x << " " << point3.y << " "
30 << point3.z << '\n';
31
32 // Point3D point4{.z = 20, .y = 1}; ERROR
33
34 needPoint({.x = 0});
35
36 std::cout << '\n';
37
38 }
```

---

Строка 20 инициализирует все члены объекта класса, но в строке 24 не указано значение для члена `x`. Поэтому `x` не проинициализирован. Вполне нормально проинициализировать только те члены, у которых нет значения по умолчанию, как в строках 28 или 34. Выражение в строке 32 не скомпилируется, поскольку `z` и `y` идут в неверном порядке.



```
x64 Native Tools Command Prompt for VS 2019
C:\Users\rainer>designatedInitializerDefaults.exe
point1: 0 1 2
point2: -1902904792 1 2
point3: 0 1 20
p: 0 1 2
C:\Users\rainer>
```

Назначенные инициализаторы со значениями по умолчанию

Назначенные инициализаторы обнаруживают преобразования в тип с меньшей точностью (сужающие преобразования), которые ведут к потере точности.

Назначенные инициализаторы обнаруживают сужающие преобразования

---

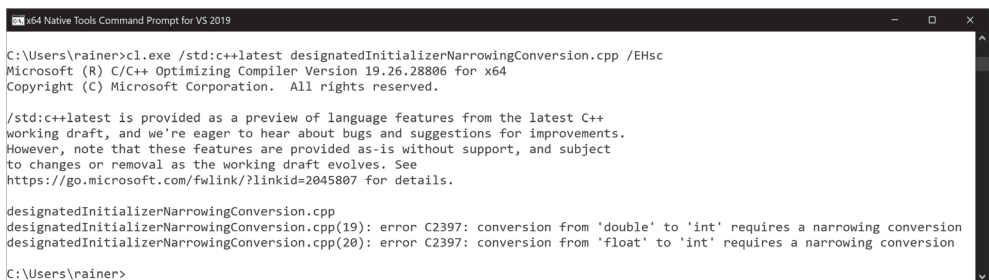
```
1 // designatedInitializerNarrowingConversion.cpp
2
3 #include <iostream>
4
5 struct Point2D{
6 int x;
7 int y;
8 };
9
```

```

10 class Point3D{
11 public:
12 int x;
13 int y;
14 int z;
15 };
16
17 int main(){
18
19 std::cout << '\n';
20
21 Point2D point2D{.x = 1, .y = 2.5};
22 Point3D point3D{.x = 1, .y = 2, .z = 3.5f};
23
24 std::cout << "point2D: " << point2D.x << " " << point2D.y << '\n';
25 std::cout << "point3D: " << point3D.x << " " << point3D.y << " "
26 << point3D.z << '\n';
27
28 std::cout << '\n';
29
30 }

```

Строки 21 и 22 приводят к ошибкам компиляции, поскольку инициализация `.y = 2.5` и `.z = 3.5f` приводит к сужающему преобразованию в `int`.



```

C:\Users\rainer>cl.exe /std:c++latest designatedInitializerNarrowingConversion.cpp /EHsc
Microsoft (R) C/C++ Optimizing Compiler Version 19.26.28806 for x64
Copyright (C) Microsoft Corporation. All rights reserved.

/std:c++latest is provided as a preview of language features from the latest C++
working draft, and we're eager to hear about bugs and suggestions for improvements.
However, note that these features are provided as-is without support, and subject
to changes or removal as the working draft evolves. See
https://go.microsoft.com/fwlink/?linkid=2045807 for details.

designatedInitializerNarrowingConversion.cpp
designatedInitializerNarrowingConversion.cpp(19): error C2397: conversion from 'double' to 'int' requires a narrowing conversion
designatedInitializerNarrowingConversion.cpp(20): error C2397: conversion from 'float' to 'int' requires a narrowing conversion
C:\Users\rainer>

```

Назначенные инициализаторы обнаруживают сужающие преобразования

Интересно, что назначенные инициализаторы в С ведут себя иначе, чем назначенные инициализаторы в С++.



### Различия между С и С++

Назначенные инициализаторы языка С поддерживают случаи использования, которые не поддерживаются в С++. В С позволено:

- ♦ инициализировать члены агрегатного объекта в произвольном порядке;
- ♦ инициализировать члены вложенного агрегатного объекта;
- ♦ смешивать назначенные инициализаторы и традиционные инициализаторы;
- ♦ возможна назначенная инициализация массивов.

В документе P0329R4<sup>1</sup> есть понятные примеры использования этих случаев.

Различие между C и C++

---

```

struct A { int x, y; };
struct B { struct A a; };
struct A a = { .y = 1, .x = 2 }; // valid C, invalid C++ (out of order)
int arr[3] = { [1] = 5 }; // valid C, invalid C++ (array)
struct B b = { .a.x = 0 }; // valid C, invalid C++ (nested)
struct A a = { .x = 1, 2 }; // valid C, invalid C++ (mixed)

```

---

Причина подобной разницы между C и C++ объясняется в этом предложении: «В C++ члены уничтожаются в обратном к созданию порядке и элементы списка инициализации выполняются в лексикографическом порядке, поэтому инициализаторы для полей должны быть заданы в таком порядке. Инициализаторы для массива конфликтуют с синтаксисом для лямбд. Вложенные инициализаторы используются редко». Далее в документе P0329R4 приводится аргументация того, что часто используемой является только инициализация с нарушениями порядка для агрегатных объектов, а остальные случаи несущественны.

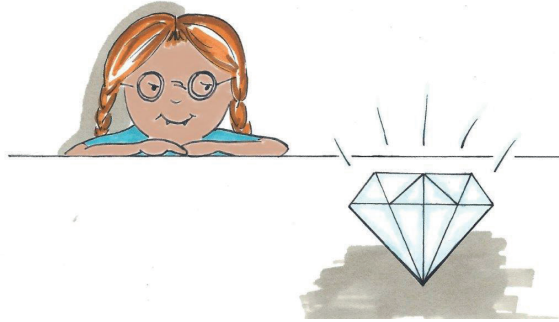


### Важные замечания

- ♦ Назначенная инициализация – это частный случай агрегатной инициализации, позволяющий инициализировать члены при помощи их имен. Порядок инициализации должен совпадать с порядком объявления.

<sup>1</sup> <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0329r4.pdf>.

## 4.5 consteval и constinit



Сиппи восхищается алмазом

С выходом стандарта C++20 мы получили два новых ключевых слова – `constexpr` и `constinit`. Ключевое слово `constexpr` создает функцию, которая выполняется во время компиляции, а `constinit` гарантирует, что переменная инициализируется во время компиляции. Сейчас у вас может возникнуть впечатление, что оба этих спецификатора крайне похожи на `constexpr`. Если коротко, то вы правы. Но прежде чем я буду сравнивать `constexpr`, `constinit`, `constexpr` и старый добрый `const`, я хотел бы подробнее рассказать о `constexpr` и `constinit`.

### 4.5.1 constexpr

Спецификатор `constexpr` создает так называемую непосредственную (immediate) функцию.

Функция `constexpr`

```
constexpr int sqr(int n) {
 return n * n;
}
```

Каждый вызов непосредственной функции создает константу времени компиляции. Говоря проще: `constexpr`-функция (непосредственная функция) выполняется во время компиляции.

Ключевое слово `constexpr` нельзя применять к деструкторам или функциям, которые что-то выделяют или освобождают в памяти. В описании вы можете использовать только один из описателей: `constexpr`, `constexpr` или `constinit`. Непосредственная функция по определению является встраиваемой и должна удовлетворять требованиям к `constexpr`-функции.

Требования, предъявляемые к `constexpr`-функциям, которые появились с вводом стандарта C++14, а следовательно, предъявляемые и к `constexpr`-функциям:

- функция `constexpr` (`constexpr`) может:
  - ◆ содержать команды условного перехода или цикла;
  - ◆ состоять из более чем одной команды;

- ♦ вызывать constexpr-функции. Функция constexpr может вызывать только constexpr-функции, но не наоборот;
- ♦ использовать базовые типы данных как переменные, которые инициализируются константными выражениями;
- функция constexpr (constexpr) не может:
  - ♦ иметь static или thread\_local данные;
  - ♦ использовать блок try или команду goto;
  - ♦ вызывать не constexpr-функции или использовать не constexpr-данные.

Говоря проще: все зависимости constexpr-функции должны быть разрешены во время компиляции.

Программа constexprSqr.cpp применяет constexpr-функцию под названием sqr.

Функция constexpr

---

```
1 // constexprSqr.cpp
2
3 #include <iostream>
4
5 constexpr int sqr(int n) {
6 return n * n;
7 }
8
9 int main() {
10
11 std::cout << "sqr(5): " << sqr(5) << '\n';
12
13 const int a = 5;
14 std::cout << "sqr(a): " << sqr(a) << '\n';
15
16 int b = 5;
17 // std::cout << "sqr(b): " << sqr(b) << '\n'; ERROR
18
19 }
```

---

Число 5 является константным выражением и может использоваться как аргумент для функции sqr (строка 11). То же самое справедливо и для переменной a (строка 13). Константная переменная, такая как a, может использоваться в константном выражении, когда она инициализируется константным выражением. Переменная b (строка 16) не является константным выражением. Поэтому вызов sqr(b) (строка 17) недопустим.

Результат работы программы следующий:

```
sqr(5): 25
sqr(a): 25
```

Использование consteval-функции

### 4.5.2 constinit

Описатель `constinit` может применяться к переменным со статическим выделением в памяти или к переменным с выделением памяти в потоке (`thread storage duration`).

- Глобальные (в пространстве имен) переменные, статические переменные или статические члены класса имеют статическое выделение памяти (`static storage duration`). Память под эти объекты выделяется тогда, когда программа начинает работу, и освобождается, когда программа завершает работу.
- Память под переменные типа `thread_local` выделяется в памяти потока. Эта память отдельно выделяется для каждого потока, который использует эти переменные. Память `thread_local` полностью принадлежит своему потоку. Соответствующие переменные создаются при первом использовании, и их время жизни привязано ко времени жизни содержащего их потока. Часто память, содержащая такие данные, называется локальной памятью потока (`thread local storage`).

Описатель `constinit` гарантирует для этих переменных, что они инициализируются во время компиляции. Описатель `constinit` не подразумевают неизменность (`constness`) переменных.

Инициализация с `constinit`

```
sqr(5): 25
sqr(a): 25
```

Переменные `res1` и `res2` имеют статическое время жизни. Переменная `res3` имеет время жизни потока.

```
sqr(5): 25
sqr(5): 25
sqr(5): 25
```

Использование инициализации `constexpr`

Теперь пора поговорить о разнице между `const`, `constexpr`, `constexpr` и `constexpr`. Сначала я разберу, как будут выполняться функции с такими спецификаторами, а потом перейду к инициализации переменных.

### 4.5.3 Выполнение функций

Следующая программа `constexpr.cpp` содержит три версии функции `square`.

Три версии функции `square`

---

```
1 // constexpr.cpp
2
3 #include <iostream>
4
5 int sqrRunTime(int n) {
6 return n * n;
7 }
8
9 constexpr int sqrCompileTime(int n) {
10 return n * n;
11 }
12
13 constexpr int sqrRunOrCompileTime(int n) {
14 return n * n;
15 }
16
17 int main() {
18
19 // constexpr int prod1 = sqrRunTime(100); ERROR
```



---

```

20 constexpr int prod2 = sqrCompileTime(100);
21 constexpr int prod3 = sqrRunOrCompileTime(100);
22
23 int x = 100;
24
25 int prod4 = sqrRunTime(x);
26 // int prod5 = sqrCompileTime(x); ERROR
27 int prod6 = sqrRunOrCompileTime(x);
28
29 }

```

---

Как подсказывает название, `sqrRunTime` (строка 5) выполняется во время выполнения, `constexpr`-функция `sqrCompileTime` (строка 9) выполняется во время компиляции, `constexpr`-функция `sqrRunOrCompileTime` может выполняться как во время компиляции, так и во время выполнения. Соответственно, если потребовать результат во время компиляции от `sqrRunTime` (строка 19), то это приведет к ошибке. Аналогично использование неконстантного выражения в качестве аргумента `sqrCompileTime` (строка 26) также является ошибкой.

Разница между `constexpr`-функцией `sqrRunOrCompileTime` и `constexpr`-функцией `sqrCompileTime` заключается в том, что `sqrRunOrCompileTime` должна быть выполнена во время компиляции, когда контекст требует выполнения ее во время компиляции.

Выполнение функции во время компиляции или во время выполнения

---

```

1 static_assert(sqrRunOrCompileTime(10) == 100); // compile time
2 int arrayNewWithConstExpressionFunction[sqrRunOrCompileTime(100)];
 // compile time
3 constexpr int prod = sqrRunOrCompileTime(100); // compile time
4
5 int a = 100;
6 int runTime = sqrRunOrCompileTime(a); // run time
7
8 int runTimeOrCompiletime = sqrRunOrCompileTime(100); // run time
9 // or compile time

```

---

Строки 1–3 требуют выполнения функции во время компиляции. Строка 6 может быть выполнена только во время выполнения, поскольку `a` – это не константное выражение. Важной строкой является строка 8. Функция может быть выполнена и во время выполнения, и во время компиляции. Будет ли она в действительности выполнена во время компиляции или же во время выполнения, зависит от компилятора и уровня оптимизации. Но на строку 10 это не распространяется. Функция типа `constexpr` всегда выполняется во время компиляции.

### 4.5.4 Инициализация переменных

Программа `constexprConstinit.cpp` сравнивает использование спецификаторов `const`, `constexpr` и `constinit`.

Сравнение `const`, `constexpr` и `constinit`

```

1 // constexprConstinit.cpp
2
3 #include <iostream>
4
5 constexpr int constexprVal = 1000;
6 constinit int constinitVal = 1000;
7
8 int incrementMe(int val){ return ++val;}
9
10 int main() {
11
12 auto val = 1000;
13 const auto res = incrementMe(val);
14 std::cout << "res: " << res << '\n';
15
16 // std::cout << "res: " << ++res << '\n'; ERROR
17 // std::cout << "++constexprVal: " << ++constexprVal << '\n'; ERROR
18 std::cout << "++constinitVal: " << ++constinitVal << '\n';
19
20 constexpr auto localConstexpr = 1000;
21 // constinit auto localConstinit = 1000; ERROR
22
23 }
```

Только переменная типа `const` (строка 13) будет проинициализирована во время выполнения. Переменные `constexpr` и `constinit` будут проинициализированы во время компиляции.

Переменная `constinit` (строка 18) не подразумевает константности, в отличие от `const` (строка 16) и `constexpr` (строка 17). Переменная, объявленная как `constexpr` (строка 20) или `const` (строка 13), в отличие от переменной `constinit` (строка 21), может быть локальной.

```

1 // constexprConstinit.cpp
2
3 #include <iostream>
4
5 constexpr int constexprVal = 1000;
6 constinit int constinitVal = 1000;
7
8 int incrementMe(int val){ return ++val;}
9
10 ...
```

Переменные `const`, `constexpr` и `constinit`

## 4.5.5 Исправляем проблему порядка статической инициализации

В соответствии с FAQ на [isocpp.org](https://isocpp.org)<sup>1</sup> «фиаско» статической инициализации (static initialization fiasco) – это «изохренный способ сломать вашу программу». Также в тексте есть такая фраза: «Порядок статической инициализации – это очень тонкий и часто неправильно понимаемый аспект C++».

Перед тем как я продолжу, я хотел бы сделать небольшую оговорку. Зависимости между переменными со статической продолжительностью жизни в различных единицах компиляции – это «дурной тон» и основание для рефакторинга. Поэтому если вы последуете этому совету и будете избегать такого стиля кодирования, то данный раздел можете пропустить.

### 4.5.5.1 «Фиаско» порядка статической инициализации

Статические переменные в одной единице компиляции инициализируются в соответствии с порядком их определения.

Для сравнения, инициализация статических переменных в разных единицах компиляции содержит в себе серьезную проблему. Когда одна статическая переменная `staticA` определена в одной единице компиляции, а другая статическая переменная `staticB` определена в другой единице компиляции, и при этом переменная `staticB` для своей инициализации требует `staticA`, то это приведет к «фиаско» порядка статической инициализации. Ваша программа будет плохо сформированной (ill-formed), поскольку у вас нет никаких гарантий, что какая-то статическая переменная будет проинициализирована первой во время выполнения.

Прежде чем я опишу, как исправить эту проблему, давайте я продемонстрирую возникновение ее на примере.

#### 4.5.5.1.1 Шанс 50:50 выполнить программу верно

Что особенного в инициализации статических переменных? Инициализация переменных происходит в два шага: статический и динамический.

Когда `static`-переменная не может быть `const`-инициализирована во время компиляции, она получает нулевое начальное значение. В последующем во время выполнения происходит динамическая инициализация для тех `static`-переменных, которые были инициализированы нулем.

«Фиаско» статической инициализации (часть 1)

---

```
// sourceSIOF1.cpp
```

```
int square(int n) {
 return n * n;
}

auto staticA = square(5);
```

---

<sup>1</sup> <https://isocpp.org/wiki/faq/ctors#static-init-order>.

«Фиаско» статической инициализации (часть 2)

---

```
1 // mainSOIF1.cpp
2
3 #include <iostream>
4
5 extern int staticA;
6 auto staticB = staticA;
7
8 int main() {
9
10 std::cout << '\n';
11
12 std::cout << "staticB: " << staticB << '\n';
13
14 std::cout << '\n';
15
16 }
```

---

В строке 5 определяется статическая переменная `staticA`. Инициализация `staticB` зависит от инициализации `staticA`. Но `staticB` инициализируется нулем во время компиляции и динамически инициализируется во время выполнения. Проблема в том, что неизвестно, в каком порядке будут инициализироваться эти статические переменные, поскольку они принадлежат разным единицам компиляции. И с вероятностью 50:50 вы получите в `staticB` значение 0 или 25.

Для демонстрации этой проблемы я изменил порядок линковки объектных файлов. В результате изменилось и значение `staticB`.

```
1 // mainSOIF1.cpp
2
3 #include <iostream>
4
5 extern int staticA;
6 auto staticB = staticA;
7
8 int main() {
9
10 std::cout << '\n';
11
12 std::cout << "staticB: " << staticB << '\n';
13
14 std::cout << '\n';
15
16 }
```

---

Это «фиаско» состоит в том, что результат выполнения зависит от порядка линковки объектных файлов. Что может сделать программист, если у него нет C++20?

#### 4.5.5.1.2 Отложенная инициализация `static` переменных с локальной видимостью

Статические переменные с локальной видимостью создаются при первом использовании. Локальная видимость означает, что статическая переменная в каком-то смысле помещена внутри фигурных скобок. Отложенная инициализация гарантируется стандартом C++98. Начиная со стандарта C++11 статические переменные с локальной видимостью также инициализируются потокобезопасным образом. Потокобезопасный паттерн Singleton Мейерса<sup>1</sup> основан на этой гарантии.

Отложенная инициализация также может использоваться для преодоления «фиаско» статической инициализации.

Отложенная инициализация `static`-переменных с локальной видимостью

---

```

1 // sourceSIOF2.cpp
2
3 int square(int n) {
4 return n * n;
5 }
6
7 int& staticA() {
8
9 static auto staticA = square(5);
10 return staticA;
11
12 }
```

---

Отложенная инициализация `static`-переменных с локальной видимостью

---

```

1 // mainSIOF2.cpp
2
3 #include <iostream>
4
5 int& staticA();
6
7 auto staticB = staticA();
8
9 int main() {
10
```

---

<sup>1</sup> [https://en.wikipedia.org/wiki/Scott\\_Meyers](https://en.wikipedia.org/wiki/Scott_Meyers).

```
11 std::cout << '\n';
12
13 std::cout << "staticB: " << staticB << '\n';
14
15 std::cout << '\n';
16
17 }
```

---

В этом случае переменная `staticA` (строка 9 в файле `sourceSIOF2.cpp`) является статической переменной с локальной областью видимости. Строка 5 в файле `mainSIOF2.cpp` объявляет функцию `staticA`, которая используется для инициализации `staticB`.

То, что переменная `staticA` имеет локальную область видимости, гарантирует, что она будет создана и проинициализирована при первом использовании. Таким образом, изменение порядка линковки не изменит значение `staticB`.



```
rainer@seminar:~$ g++ -c mainSIOF2.cpp
rainer@seminar:~$ g++ -c sourceSIOF2.cpp
rainer@seminar:~$ g++ mainSIOF2.o sourceSIOF2.o -o mainSource
rainer@seminar:~$ g++ sourceSIOF2.o mainSIOF2.o -o sourceMain
rainer@seminar:~$ mainSource

staticB: 25

rainer@seminar:~$ sourceMain

staticB: 25

rainer@seminar:~$
```

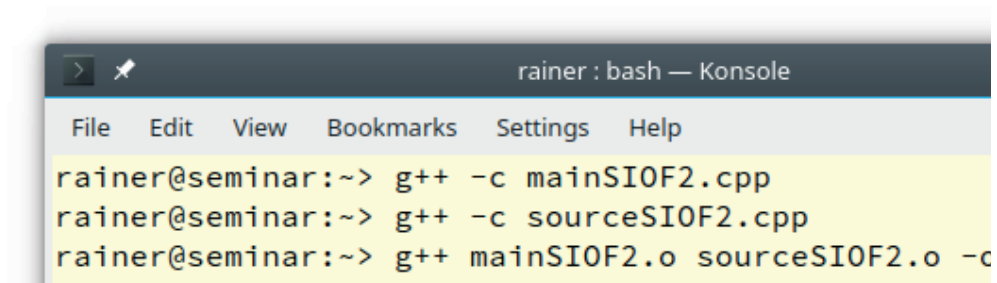
Решение проблемы «фиаско» статической инициализации при помощи статических переменных с локальной видимостью

На заключительном шаге я исправляю «фиаско» статической инициализации при помощи C++20.

#### 4.5.5.1.3 Инициализация статической переменной во время компиляции

Давайте применим `constexpr` к `staticA`. `constexpr` гарантирует, что `staticA` будет проинициализирована во время компиляции.

Инициализация статической переменной во время компиляции (часть 1)



```
rainer@seminar:~> g++ -c mainSIOF2.cpp
rainer@seminar:~> g++ -c sourceSIOF2.cpp
rainer@seminar:~> g++ mainSIOF2.o sourceSIOF2.o -c
```

Инициализация статической переменной во время компиляции (часть 2)

```
1 // mainSIOF3.cpp
2
3 #include <iostream>
4
5 extern constinit int staticA;
6
7 auto staticB = staticA;
8
9 int main() {
10
11 std::cout << '\n';
12
13 std::cout << "staticB: " << staticB << '\n';
14
15 std::cout << '\n';
16
17 }
```

Строка 5 файла mainSIOF3.cpp объявляет переменную staticA, которая инициализируется (строка 7 файла sourceSIOF3.cpp) во время компиляции. Обратите внимание, что использование constexpr (строка 5 файла mainSIOF3.cpp) вместо constinit будет ошибкой, поскольку constexpr требует определения, а не просто объявления.



```
Windows PowerShell
C:\Users\rainer>clang++ -std=c++20 -c mainSIOf3.cpp

C:\Users\rainer>clang++ -std=c++20 -c sourceSIOf3.cpp

C:\Users\rainer>clang++ mainSIOf3.o sourceSIOf3.o -o mainSource.exe

C:\Users\rainer>clang++ sourceSIOf3.o mainSIOf3.o -o sourceMain.exe

C:\Users\rainer>mainSource.exe

staticB: 25

C:\Users\rainer>sourceMain.exe

staticB: 25

C:\Users\rainer>
```

Решение «фиаско» со статической инициализацией при помощи `constinit`

Как и в случае с отложенной инициализацией локальной статической переменной, мы получим значение `staticB`, равное 25.

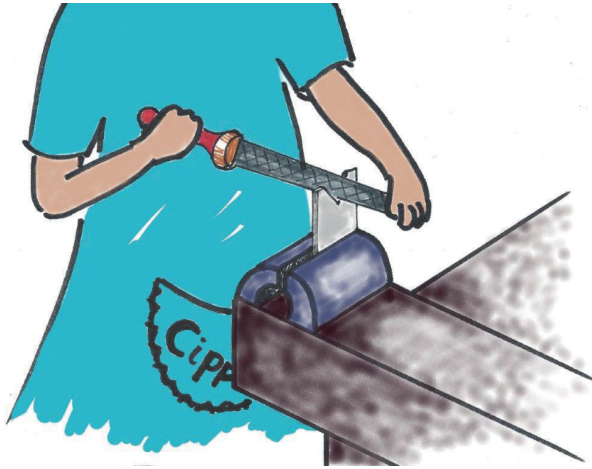


### Важные замечания

- ♦ В C++20 мы получили два новых ключевых слова – `constexpr` и `constinit`. `constexpr` создает функцию, которая будет выполняться во время компиляции, а `constinit` гарантирует, что соответствующая переменная будет проинициализирована во время компиляции.
- ♦ В отличие от `constexpr` в стандарте C++11, `constexpr` гарантирует, что функция будет выполняться во время компиляции.
- ♦ Есть тонкая разница между `const`, `constexpr`, `constexpr` и `constinit`. Описатели `const` и `constexpr` создают константные переменные, а `constexpr` и `constinit` выполняются во время компиляции.



## 4.6 Улучшение работы с шаблонами



Сиппи пользуется новыми инструментами

Улучшения работы с шаблонами делают разработку программ на C++20 более последовательной и, следовательно, менее подверженной ошибкам при использовании обобщенного программирования.

### 4.6.1 Условный явный конструктор

Иногда вам нужен класс, который должен иметь конструкторы, принимающие различные типы. Например, у вас есть класс `VariantWrapper`, который содержит внутри себя `std::variant`, принимающий различные типы.

Класс `VariantWrapper` с атрибутом `std::variant`

---

```
class VariantWrapper {

 std::variant<bool, char, int, double, float, std::string> myVariant;

};
```

---

Для инициализации `VariantWrapper` при помощи `bool`, `char`, `int`, `double`, `float` или `std::string` этому классу нужен конструктор для каждого из перечисленных типов. Лень бывает полезна – по крайней мере для программистов, – вы решаете сделать конструктор обобщенным.

Класс `Implicit` демонстрирует обобщенный конструктор.

Обобщенный конструктор

---

```
1 // implicitExplicitGenericConstructor.cpp
2
3 #include <iostream>
4 #include <string>
5
6 struct Implicit {
7 template <typename T>
8 Implicit(T t) {
9 std::cout << t << '\n';
10 }
11 };
12
13 struct Explicit {
14 template <typename T>
15 explicit Explicit(T t) {
16 std::cout << t << '\n';
17 }
18 };
19
20 int main() {
21
22 std::cout << '\n';
23
24 Implicit imp1 = "implicit";
25 Implicit imp2("explicit");
26 Implicit imp3 = 1998;
27 Implicit imp4(1998);
28
29 std::cout << '\n';
30
31 // Explicit exp1 = "implicit";
32 Explicit exp2{"explicit"};
33 // Explicit exp3 = 2011;
34 Explicit exp4{2011};
35
36 std::cout << '\n';
37
38 }
```

---

Теперь возникла проблема. Обобщенный конструктор (строка 7) может быть вызван с любым типом. Он слишком «жадный». Поместив перед конструктором слово `explicit` (строка 14), неявные преобразования (строки 31 и 33) станут недействительными. Только явные вызовы (строки 32 и 34) являются действительными.

```
implicit
explicit
1998
1998

explicit
2011
```

Явные и неявные обобщенные конструкторы

В стандарте C++20 `explicit` оказывается еще более полезным. Представьте себе, что у вас есть тип `MyBool`, который поддерживает только неявные преобразования из `bool`, но больше никаких других неявных преобразований нет. В этом случае можно использовать `explicit` условным образом.

Обобщенный конструктор, который позволяет делать неявные преобразования из `bool`

---

```
1 // conditionallyConstructor.cpp
2
3 #include <iostream>
4 #include <type_traits>
5 #include <typeinfo>
6
7 struct MyBool {
8 template <typename T>
9 explicit(!std::is_same<T, bool>::value) MyBool(T t) {
10 std::cout << typeid(t).name() << '\n';
11 }
12 };
13
14 void needBool(MyBool b){ }
15
16 int main() {
17
18 MyBool myBool1(true);
19 MyBool myBool2 = false;
20
```

```
21 needBool(myBool1);
22 needBool(true);
23 // needBool(5);
24 // needBool("true");
25
26 }
```

---

Выражение `explicit(!std::is_same<T, bool>::value)` гарантирует, что переменная типа `MyBool` может быть создана из `bool` только неявным способом. Функция `std::is_same` – предикат времени компиляции из библиотеки `type_traits`. Такой предикат, как `std::is_same`, выполняется во время компиляции и возвращает значение `boolean`. Следовательно, неявные преобразования из `bool` (строки 19 и 22) возможны, а вот закомментированные строки 23 и 24, где происходит преобразование из `int` и `C-string`, – нет.

## 4.6.2 Нетипизированные параметры шаблона

C++ поддерживает параметры шаблона, которые не являются типизированными. По сути, такими параметрами могут быть:

- целые числа и перечисления;
- указатели или ссылки на объекты, функции и атрибуты класса;
- `std::nullptr_t`.



### Типичные нетипизированные параметры

Когда я спрашиваю студентов в своем классе, использовали ли они когда-нибудь параметр шаблона, не являющийся типизированным, они говорят: «Нет!». Тогда я обычно привожу часто встречающийся пример параметра, не являющегося типом.

Определение `std::array`

---

```
std::array<int, 5> myVec;
```

---

Константа 5 в этом примере используется как аргумент шаблона.

С самого первого стандарта C++ – C++98 – шло постоянное обсуждение в сообществе C++ поддержки чисел с плавающей точкой в качестве параметра шаблона. Теперь это реализовано, и даже более: C++20 поддерживает числа с плавающей точкой, литеральные типы и строковые литералы как параметры, не являющиеся типизированными.

### 4.6.2.1 Числа с плавающей точкой и литеральные типы

Литеральные типы обладают следующими основными свойствами:

- все базовые классы и нестатические члены являются публичными и неизменяемыми;
- типы всех базовых классов и нестатических членов данных являются структурами, массивами или же построены из них.

Литеральный тип должен обладать constexpr-конструктором. Следующая программа использует числа с плавающей точкой и литеральные значения в качестве параметров шаблона.

Числа с плавающей точкой и литеральные значения в качестве параметров шаблона

---

```

1 // nonTypeTemplateParameter.cpp
2
3 struct ClassType {
4 constexpr ClassType(int) {}
5 };
6
7 template <ClassType c1>
8 auto getClassType() {
9 return c1;
10 }
11
12 template <double d>
13 auto getDouble() {
14 return d;
15 }
16
17 int main() {
18
19 auto c1 = getClassType<ClassType(2020)>();
20
21 auto d1 = getDouble<5.5>();
22 auto d2 = getDouble<6.5>();
23
24 }
```

---

Класс ClassType обладает constexpr-конструктором (строка 4), и поэтому его экземпляры могут быть использованы в качестве аргумента шаблона (строка 9). То же самое относится и к шаблонной функции getDouble (строка 13), которая принимает только значения типа double. Я хотел бы подчеркнуть, что каждый вызов шаблонной функции getDouble (строки 21 и 22) создает свой экземпляр getDouble. Эта функция будет полностью специализирована под заданный параметр шаблона.

Начиная с введения стандарта C++20 строки могут быть использованы как аргументы шаблона, не являющиеся типизированными.

#### 4.6.2.2 Строковые литералы

У класса StringLiteral имеется constexpr-конструктор.

Строковые литералы как параметры шаблонов

---

```
1 // nonTypeTemplateParameterString.cpp
2
3 #include <algorithm>
4 #include <iostream>
5
6 template <int N>
7 class StringLiteral {
8 public:
9 constexpr StringLiteral(char const (&str)[N]) {
10 std::copy(str, str + N, data);
11 }
12 char data[N];
13 };
14
15 template <StringLiteral str>
16 class ClassTemplate {};
17
18 template <StringLiteral str>
19 void FunctionTemplate() {
20 std::cout << str.data << '\n';
21 }
22
23 int main() {
24
25 std::cout << '\n';
26
27 ClassTemplate<"string literal"> cls;
28 FunctionTemplate<"string literal">();
29
30 std::cout << '\n';
31
32 }
```

---

StringLiteral – это литеральный тип, поэтому значения этого типа могут быть использованы в качестве параметров для ClassTemplate (строка 15) и FunctionTemplate (строка 18). Конструктор constexpr принимает C-string в качестве аргумента.

```
3 #include <algorithm>
4 #include <iostream>
5 template <int N>
```

Вы можете удивиться, зачем нам нужны строковые литералы как параметры шаблона? Ответ ниже.



### Регулярные выражения времени компиляции

Очень впечатляющим примером использования строковых литералов являются регулярные выражения времени компиляции<sup>1</sup>. Уже есть предложение по их включению в C++23 – P1433R0 Compile-Time Regular Expressions<sup>2</sup>. Хана Дусикова, автор данного предложения, пишет следующее: «Текущие дизайн и реализация `std::regex` в C++ (библиотека регулярных выражений<sup>3</sup>) являются в основном медленными, поскольку регулярные выражения разбираются и компилируются во время выполнения. Пользователям часто не нужен разбор регулярных выражений во время выполнения, поскольку во многих случаях само выражение уже известно во время компиляции. Я полагаю, что это нарушает философию C++, выраженную в следующей фразе: “не плати за то, что ты не используешь”».

Если регулярные выражения известны во время компиляции, то шаблон должен быть проверен во время компиляции. Дизайн `std::regex` этого не поддерживает, так как для него входом являются строки времени выполнения и синтаксические ошибки сообщаются как исключения».



### Важные замечания

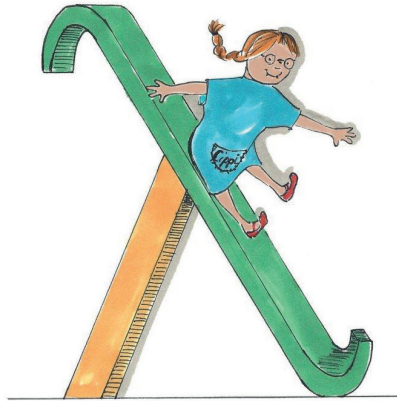
- ♦ Условный явный конструктор позволяет явно управлять тем, какие типы могут быть использованы в обобщенном конструкторе.
- ♦ C++20 поддерживает числа с плавающей точкой как параметры шаблона, не являющиеся типизированными.

<sup>1</sup> <https://github.com/hanickadot/compile-time-regular-expressions>.

<sup>2</sup> <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1433r0.pdf>.

<sup>3</sup> <https://en.cppreference.com/w/cpp/regex>.

## 4.7 Улучшения лямбд



Сиппи скатывается по дорожке

В стандарте C++20 лямбды поддерживают шаблонные параметры и, следовательно, концепты, обладают конструктором по умолчанию и поддерживают присваивание, когда у них нет состояния. Кроме того, лямбда-выражения могут быть использованы в контекстах, не требующих выполнения (unevaluated context). Начиная с C++20 лямбды сами замечают, когда вы неявно копируете указатель `this`. Все это значит, что значительная часть неопределенного поведения, связанного с лямбдами, теперь в прошлом.

Давайте начнем с шаблонных параметров для лямбд.

### 4.7.1 Шаблонные параметры для лямбд

На самом деле разница между лямбдами с типом (C++11), обобщенными лямбдами (C++14) и шаблонными лямбдами (шаблонный параметр в лямбде) в C++20 очень незначительна.

Типизированные лямбды, обобщенные лямбды и шаблонные лямбды

---

```
1 // templateLambda.cpp
2
3 #include <iostream>
4 #include <string>
5 #include <vector>
6
7 auto sumInt = [](int fir, int sec) { return fir + sec; };
8 auto sumGen = [](auto fir, auto sec) { return fir + sec; };
9 auto sumDec = [](auto fir, decltype(fir) sec) { return fir + sec; };
10 auto sumTem = [<typename T>(T fir, T sec) { return fir + sec; };
11
12 int main() {
```



```

13
14 std::cout << '\n';
15
16 std::cout << "sumInt(2000, 11): " << sumInt(2000, 11) << '\n';
17 std::cout << "sumGen(2000, 11): " << sumGen(2000, 11) << '\n';
18 std::cout << "sumDec(2000, 11): " << sumDec(2000, 11) << '\n';
19 std::cout << "sumTem(2000, 11): " << sumTem(2000, 11) << '\n';
20
21 std::cout << '\n';
22
23 std::string hello = "Hello ";
24 std::string world = "world";
25 //std::cout<<"sumInt(hello,world):"<<sumInt(hello,world)<<'\n';
26 std::cout << "sumGen(hello, world): " << sumGen(hello, world) << '\n';
27 std::cout << "sumDec(hello, world): " << sumDec(hello, world) << '\n';
28 std::cout << "sumTem(hello, world): " << sumTem(hello, world) << '\n';
29
30
31 std::cout << '\n';
32
33 std::cout << "sumInt(true, 2010): " << sumInt(true, 2010) << '\n';
34 std::cout << "sumGen(true, 2010): " << sumGen(true, 2010) << '\n';
35 std::cout << "sumDec(true, 2010): " << sumDec(true, 2010) << '\n';
36 // std::cout << "sumTem(true, 2010): " << sumTem(true, 2010) << '\n';
37
38 std::cout << '\n';
39
40 }

```

Прежде чем я покажу довольно удивительный вывод этой программы, я хотел бы сравнить четыре лямбды.

- `sumInt`
  - ◆ C++ 11
  - ◆ Типизированная лямбда
  - ◆ Принимает только типы, переводимые в `int`
- `sumGen`
  - ◆ C++ 14
  - ◆ Обобщенная лямбда
  - ◆ Принимает все типы
- `sumTem`
  - ◆ C++ 20
  - ◆ Шаблонная лямбда
  - ◆ Первый тип и второй типы должны совпадать

- `sumDec`
  - ◆ C++ 20
  - ◆ Обобщенная лямбда
  - ◆ Второй тип должен быть конвертируемым в первый тип

Что это значит для шаблонных аргументов различных типов? Ответ: каждая лямбда принимает `int` (строки 16–19), а типизированная лямбда `sumInt` не принимает строки (строка 25).

Вызов лямбды с `bool true` и `int 2010` может быть неожиданным (строки 33–36).

- `sumInt` возвращает 2011, поскольку `true` – это целочисленное значение, переводимое в `int`.
- `sumGen` возвращает 2011, поскольку `true` – это целочисленное значение, переводимое в `int`. Но есть тонкая разница между `sumInt` и `sumGen`, которую я покажу буквально через несколько строк.
- `sumDec` возвращает 2. Почему? Тип второго параметра `sec` становится типом первого параметра `fir`: благодаря `decltype(fir) sec` компилятор выводит тип `fir` и строит по нему тип `sec`. Соответственно, 2010 переводится в `true`. В выражении `fir+sec` логическое значение `true` превращается в целочисленное значение 1. Окончательный результат 2.
- `sumTem` не допустим.

```
sumInt(2000, 11): 2011
sumGen(2000, 11): 2011
sumDec(2000, 11): 2011
sumTem(2000, 11): 2011

sumGen(hello, world): Hello world
sumDec(hello, world): Hello world
sumTem(hello, world): Hello world

sumInt(true, 2010): 2011
sumGen(true, 2010): 2011
sumDec(true, 2010): 2
```

Тонкая разница между типизированными, обобщенными и шаблонными лямбдами

Более типичным примером использования шаблонных лямбд является применение в лямбдах контейнеров. В следующей программе представлены три лямбды, принимающие на вход контейнер. Каждая лямбда возвращает размер контейнера.

Три лямбды, принимающие на вход контейнер

---

```
1 // templateLambdaVector.cpp
2
3 #include <concepts>
4 #include <deque>
5 #include <iostream>
```

---

```

6 #include <string>
7 #include <vector>
8
9 auto lambdaGeneric = [](const auto& container) { return container.size(); };
10 auto lambdaVector = [<typename T>(const std::vector<T>& vec) { return vec.size(); }];
11 auto lambdaVectorIntegral = [<std::integral T>(const std::vector<T>& vec) {
12 return vec.size();
13 }];
14
15 int main() {
16
17
18 std::cout << '\n';
19
20 std::deque deq{1, 2, 3};
21 std::vector vecDouble{1.1, 2.2, 3.3, 4.4};
22 std::vector vecInt{1, 2, 3, 4, 5};
23
24 std::cout << "lambdaGeneric(deq): " << lambdaGeneric(deq) << '\n';
25 // std::cout << "lambdaVector(deq): " << lambdaVector(deq) << '\n';
26 // std::cout << "lambdaVectorIntegral(deq): "
27 // << lambdaVectorIntegral(deq) << '\n';
28
29 std::cout << '\n';
30
31 std::cout << "lambdaGeneric(vecDouble): " << lambdaGeneric(vecDouble) << '\n';
32 std::cout << "lambdaVector(vecDouble): " << lambdaVector(vecDouble) << '\n';
33 // std::cout << "lambdaVectorIntegral(vecDouble): "
34 // << lambdaVectorIntegral(vecDouble) << '\n';
35
36 std::cout << '\n';
37
38 std::cout << "lambdaGeneric(vecInt): " << lambdaGeneric(vecInt) << '\n';
39 std::cout << "lambdaVector(vecInt): " << lambdaVector(vecInt) << '\n';
40 std::cout << "lambdaVectorIntegral(vecInt): "
41 << lambdaVectorIntegral(vecInt) << '\n';
42
43 std::cout << '\n';
44
45 }

```

---

Функция `lambdaGeneric` (строка 9) может быть вызвана с любым типом данных, у которого есть метод `size()`. Функция `lambdaVector` (строка 10) более конкретна: она принимает только `std::vector`. Функция `lambdaVectorIntegral` (строка 11) использует концепт из C++20 `std::integral`. Поэтому она принимает лишь `std::vector` значений целочисленного типа, например `int`. Для использования концепта `std::integral` мне нужно было подключить заголовочный файл `<concepts>`. Я считаю, что эта маленькая программа вполне показательна.

```

lambdaGeneric(deq): 3

lambdaGeneric(vecDouble): 4
lambdaVector(vecDouble): 4

lambdaGeneric(vecInt): 5
lambdaVector(vecInt): 5
lambdaVectorIntegral(vecInt): 5

```

Лямбды, принимающие контейнер и `std::vector`



### Вывод аргумента шаблонного класса

Есть одна особенность в программе `templateLambdaVector.cpp`, которую вы, возможно, пропустили. Начиная со стандарта C++17 компилятор может вывести тип шаблонного класса из его аргументов (строки 20–22). Поэтому вместо длинного выражения `std::vector<int> myVec {1, 2, 3}` вы можете просто написать `std::vector myVec { 1, 2, 3 }`.

## 4.7.2 Определение неявного копирования указателя `this`

В стандарте C++20 компилятор определяет, когда вы неявно копируете указатель `this`. Неявный захват указателя `this` может привести к неопределенному поведению. Неопределенное поведение означает, что нет никаких гарантий по поведению программы, такой как, например, приведена далее.

Неявный захват указателя `this` через копирование

```

1 // lambdaCaptureThis.cpp
2
3 #include <iostream>
4 #include <string>
5
6 struct LambdaFactory {
7 auto foo() const {
8 return [=] { std::cout << s << '\n'; };
9 }
10 std::string s = "LambdaFactory";
11 ~LambdaFactory() {
12 std::cout << "Goodbye" << '\n';
13 }
14 };
15

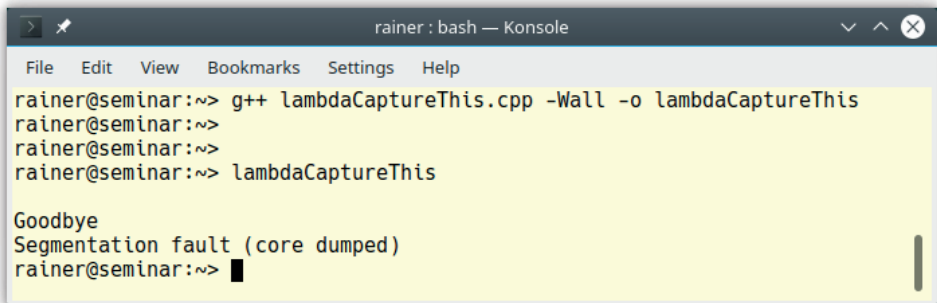
```

```

16 auto makeLambda() {
17 LambdaFactory lambdaFactory;
18
19 return lambdaFactory.foo();
20 }
21
22
23 int main() {
24
25 std::cout << '\n';
26
27 auto lam = makeLambda();
28 lam();
29
30 std::cout << '\n';
31
32 }

```

Компиляция этой программы происходит. Но выполнение программы приводит к ошибкам.



```

rainer@seminar:~$ g++ lambdaCaptureThis.cpp -Wall -o lambdaCaptureThis
rainer@seminar:~$./lambdaCaptureThis
Goodbye
Segmentation fault (core dumped)
rainer@seminar:~$

```

Ошибка сегментации (segmentation fault) из-за неопределенного поведения программы

Почему выполнение программы `lambdaCaptureThis.cpp` приводит к ошибкам? Метод `foo` (строка 7) возвращает `lambda [=] { std::cout << s << '\n'; }`, выполняя неявное копирование указателя `this`. Это неявное копирование не приводит к ошибкам в строке 17, но вызывает ошибку при выходе за пределы области видимости программы. Выход за пределы области видимости означает конец жизни локальной лямбды (строка 19). Поэтому вызов `lam()` (строка 29) приводит к неопределенному поведению.

Компилятор C++20 должен в этом случае выдать предупреждение.

```
<source>:8:16: warning: implicit capture of 'this' via '[' is deprecated in C++20 [-Wdeprecated]
 8 | return [=] { std::cout << s << std::endl; };
 | ^
<source>:8:16: note: add explicit 'this' or '*this' capture
Execution build compiler returned: 0
Program returned: 139

Goodbye
```

Предупреждение, выданное компилятором C++20

Последние две возможности лямбд в C++20 крайне удобны, когда вы их комбинируете: лямбды в C++20 могут быть созданы при помощи конструктора по умолчанию и поддерживают оператор копирования, когда у них нет состояния. Кроме того, лямбды могут быть использованы в контекстах без их выполнения.

### 4.7.3 Лямбды в контекстах без выполнения. Использование конструктора по умолчанию и копирования для лямбд без состояния

Этот раздел может содержать два новых для вас понятия: контекст, не требующий выполнения, и лямбды без состояния.

#### 4.7.3.1 Контекст без выполнения

Следующий пример кода содержит объявление функции и определение функции.

Объявление и определение функции

```
int add1(int, int); // declaration
int add2(int a, int b) { return a + b; } // definition
```

Функция `add1` объявлена, а функция `add2` определена. Это значит, что если вы используете `add1` в контексте, требующем выполнения, например вызовете ее, то вы получите ошибку линковки. Но важным фактором является то, что вы можете использовать `add1` в контекстах, не требующих выполнения, например `typeid`<sup>1</sup> или `decltype`<sup>2</sup>. Оба этих оператора принимают операнды, не требуя их выполнения.

Контекст без выполнения

```
1 // unevaluatedContext.cpp
2
3 #include <iostream>
4 #include <typeinfo> // typeid
5
6 int add1(int, int); // declaration
7 int add2(int a, int b) { return a + b; } // definition
8
```

<sup>1</sup> <https://en.cppreference.com/w/cpp/language/typeid>.

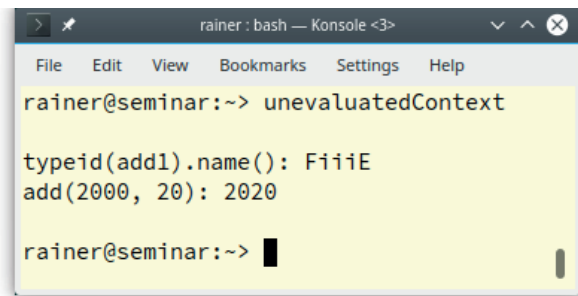
<sup>2</sup> <https://en.cppreference.com/w/cpp/language/decltype>.

```

9 int main() {
10
11 std::cout << '\n';
12
13 std::cout << "typeid(add1).name(): " << typeid(add1).name() << '\n';
14
15 decltype(*add1) add = add2;
16
17 std::cout << "add(2000, 20): " << add(2000, 20) << '\n';
18
19 std::cout << '\n';
20
21 }

```

Выражение `typeid(add1).name()` (строка 13) возвращает строковое представление типа. А выражение `decltype` (строка 15) выводит тип своего аргумента.



Использование контекста без выполнения

#### 4.7.3.2 Лямбды без состояния

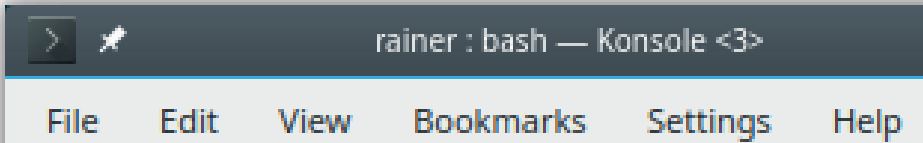
Лямбда без состояния – это лямбда, которая ничего не захватывает. Или, говоря иначе, лямбда без состояния – это лямбда, где квадратные скобки `[]` в самом начале ничего не содержат. Например, следующая лямбда `auto add = [](int a, int b) { return a+b; }` является лямбдой без состояния.

#### 4.7.3.3 Адаптивные ассоциативные контейнеры из стандартной библиотеки шаблонов

Прежде чем я покажу вам следующий пример, мне хотелось бы сделать несколько замечаний. Контейнер `std::set` и все остальные упорядоченные ассоциативные контейнеры из стандартной библиотеки шаблонов (`std::map`, `std::multiset` и `std::multimap`) по определению используют функциональный объект `std::less` для сортировки ключей. `std::less` сортирует все ключи лексикографически в порядке возрастания. Объявление `std::set`<sup>1</sup> неявно использует `std::less`.

<sup>1</sup> <https://en.cppreference.com/w/cpp/container/set>.

Объявление `std::set`



Теперь давайте немного разберемся с упорядочиванием.

Использование лямбд в контексте без выполнения

```

1 // lambdaUnevaluatedContext.cpp
2
3 #include <cmath>
4 #include <iostream>
5 #include <memory>
6 #include <set>
7 #include <string>
8
9 template <typename Cont>
10 void printContainer(const Cont& cont) {
11 for (const auto& c: cont) std::cout << c << " ";
12 std::cout << "\n";
13 }
14
15 int main() {
16
17 std::cout << '\n';
18
19 std::set<std::string> set1 = {"scott", "Bjarne", "Herb", "Dave", "michael"};
20 printContainer(set1);
21
22 using SetDecreasing = std::set<std::string,
23 decltype([](const auto& l, const auto& r) {
24 return l > r;
25 })>;
26 SetDecreasing set2 = {"scott", "Bjarne", "Herb", "Dave", "michael"};
27 printContainer(set2);
28
29 using SetLength = std::set<std::string,
30 decltype([](const auto& l, const auto& r) {
31 return l.size() < r.size();
32 })>;
33 SetLength set3 = {"scott", "Bjarne", "Herb", "Dave", "michael"};
34 printContainer(set3);
35

```



```

36 std::cout << '\n';
37
38 std::set<int> set4 = {-10, 5, 3, 100, 0, -25};
39 printContainer(set4);
40
41 using setAbsolute = std::set<int, decltype([](const auto& l, const auto& r) {
42 return std::abs(l) < std::abs(r);
43 })>;
44 setAbsolute set5 = {-10, 5, 3, 100, 0, -25};
45 printContainer(set5);
46
47 std::cout << "\n\n";
48
49 }

```

Контейнеры `set1` (строка 19) и `set4` (строка 38) сортируют свои ключи в порядке возрастания. Каждый из контейнеров `set2` (строка 26), `set3` (строка 33) и `set5` (строка 44) сортирует свои ключи своим особенным образом, используя лямбды в контексте без выполнения. Ключевое слово `using` (строка 22) определяет альтернативное имя для типа, который используется далее (строка 26) для определения множеств. Объявление `std::set` приводит к вызову конструктора по умолчанию для лямбды без состояния.

Ниже приводится вывод этой программы.

```

1 // lambdaUnevaluatedContext.cpp
2
3 #include <cmath>
4 #include <iostream>
5 #include <memory>
6 #include <set>
7 #include <string>
8
9 template <typename Cont>
10 void printContainer(const Cont& cont) {
11 for (const auto& c: cont) std::cout << c << " ";
12 std::cout << "\n";
13 }
14

```

Использование лямбды в контексте без выполнения

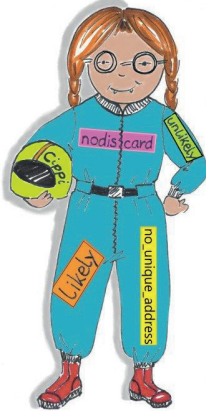
Вывод программы может вас удивить. Множество `set3`, использующее лямбду `[](const auto& l, const auto& r) { return l.size () < r.size (); }` в качестве предиката, игнорирует строку `Dave`. Причина этого очень проста. У строки `Dave` та же длина, что и у `Herb`, которая была добавлена ранее. `std::set` поддерживает уникальные ключи, а эти две строки идентичны в смысле используемого специального предиката. Если бы я использовал `std::multiset`, то этого бы не случилось.



#### Важные замечания

- ♦ В C++20 лямбды могут иметь шаблонные параметры. Кроме того, лямбды сами определяют, когда неявно используется указатель `this`.

## 4.8 Новые атрибуты



Сиппи готовится к гонке

В стандарте C++20 мы получили новые и улучшенные атрибуты, такие как `[[nodiscard("reason")]]`, `[[likely]]`, `[[unlikely]]` и `[[no_unique_address]]`. В частности, `[[nodiscard("reason")]]` может использоваться для явного задания намерения интерфейса.



### Атрибуты

Атрибуты позволяют программистам накладывать дополнительные ограничения на исходный код и давать компилятору новые возможности для оптимизации. Вы можете использовать атрибуты для типов, переменных, функций, имен и блоков кода. Когда вы используете более одного атрибута, вы можете применять их один за другим (`func1`) или все вместе в одном атрибуте, разделяя запятыми (`func2`).

Использование атрибутов

```
1 [[attribute1]] [[attribute2]] [[attribute3]]
2 int func1();
3
4 [[attribute1, attribute2, attribute3]]
5 int func2();
```

Атрибуты могут быть зависимыми от реализации расширениями языка или стандартными атрибутами, такими как следующие атрибуты, которые появились в стандартах C++11–C++17:

- `[[noreturn]]` (C++11): обозначает, что функция не возвращает значений;
- `[[carries_dependency]]` (C++11): обозначает цепочку зависимостей в упорядочении `release-consume`<sup>1</sup>;

<sup>1</sup> [https://en.cppreference.com/w/cpp/atomic/memory\\_order#Release-Consume\\_ordering](https://en.cppreference.com/w/cpp/atomic/memory_order#Release-Consume_ordering).

- ♦ `[[deprecated]]` (C++14): означает, что вы не должны использовать это имя;
- ♦ `[[fallthrough]]` (C++17): означает, что сквозное прохождение через `case` в операторе `switch` намеренное;
- ♦ `[[maybe_unused]]` (C++17): подавляет предупреждение компилятора о неиспользуемом имени.

### 4.8.1 `[[nodiscard("reason")]]`

В стандарте C++17 введен новый атрибут `[[nodiscard]]`. Но в нем нельзя было указать причину его вызова. В C++20 была добавлена возможность добавлять к этому атрибуту сообщения.

Отбрасывание значений и объектов

---

```

1 // withoutNodiscard.cpp
2
3 #include <utility>
4
5 struct MyType {
6
7 MyType(int, bool) {}
8
9 };
10
11 template <typename T, typename ... Args>
12 T* create(Args&& ... args) {
13 return new T(std::forward<Args>(args)...);
14 }
15
16 enum class ErrorCode {
17 Okay,
18 Warning,
19 Critical,
20 Fatal
21 };
22
23 ErrorCode errorProneFunction() { return ErrorCode::Fatal; }
24
25 int main() {
26
27 int* val = create<int>(5);
28 delete val;

```

```
29
30 create<int>(5);
31
32 errorProneFunction();
33
34 MyType(5, true);
35
36 }
```

---

Фабричная функция `create` (строка 11) может вызывать любой конструктор и возвращать объект, выделенный в куче (общей памяти программ).

У этой программы есть много потенциальных проблем. Во-первых, в строке 30 есть утечка памяти, поскольку `int` создается в общей памяти и не уничтожается. Во-вторых, код ошибки из функции `errorProneFunction` (строка 32) не проверяется. И наконец, конструктор `MyType(5, true)` (строка 34) создает временный объект, который создается и сразу же уничтожается. Это, как минимум, является тратой ресурсов. Именно здесь атрибут `[[nodiscard]]` приходит на помощь.

Атрибут `[[nodiscard]]` может использоваться в объявлении функции, определении перечисления или описании класса. Если вы отбрасываете значение функции, объявленной как `[[nodiscard]]`, то компилятор должен выдать предупреждение. То же самое относится и к функции, возвращающей перечисления или класс, объявленные как `[[nodiscard]]`, через копирование.

Давайте посмотрим, что это значит. В следующем примере я использовал синтаксис C++17 для атрибута `[[nodiscard]]`.

Использование атрибута `[[nodiscard]]` из C++17

---

```
1 // nodiscard.cpp
2
3 #include <utility>
4
5 struct MyType {
6
7 MyType(int, bool) {}
8
9 };
10
11 template <typename T, typename ... Args>
12 [[nodiscard]]
13 T* create(Args&& ... args){
14 return new T(std::forward<Args>(args)...);
15 }
16
```

```

17 enum class [[nodiscard]] ErrorCode {
18 Okay,
19 Warning,
20 Critical,
21 Fatal
22 };
23
24 ErrorCode errorProneFunction() { return ErrorCode::Fatal; }
25
26 int main() {
27
28 int* val = create<int>(5);
29 delete val;
30
31 create<int>(5);
32
33 errorProneFunction();
34
35 MyType(5, true);
36
37 }

```

Фабричная функция `create` (строка 13) и `enum ErrorCode` (строка 17) объявлены как `[[nodiscard]]`. Поэтому мы получаем предупреждения при выполнении строк 31 и 33.

```

rainer@seminar:~$ g++ nodiscard.cpp -o nodiscard
nodiscard.cpp:13:16: warning: ignoring return value of 'T* create(Arg5&& ...) [with T = int; Args = {int}]', declared with attribute nodiscard [-Wunused-result]
 create<int>(5);
    ~~~~~^~~~~
nodiscard.cpp:13:4: note: declared here
    T* create(Arg5&& ... args){
    ~~~~~^~~~~
nodiscard.cpp:33:23: warning: ignoring return value of type 'ErrorCode', declared with attribute nodiscard [-Wunused-result]
 errorProneFunction(); // (2)
    ~~~~~^~~~~
nodiscard.cpp:24:11: note: in call to 'ErrorCode errorProneFunction()', declared here
    ErrorCode errorProneFunction() { return ErrorCode::Fatal; }
    ~~~~~^~~~~
nodiscard.cpp:17:26: note: 'ErrorCode' declared here
 enum class [[nodiscard]] ErrorCode {
    ~~~~~^~~~~
rainer@seminar:~$

```

Компилятор C++17 выдает предупреждение об отброшенном объекте и коде ошибки

Программа стала гораздо лучше, но у нее все еще есть проблемы. `[[nodiscard]]` нельзя использовать для таких функций, как конструктор, которые ничего не возвращают. Поэтому временный объект `MyType(5, true)` (строка 35) по-прежнему создается без предупреждений. Во-вторых, сообщения об ошибках слишком общие. Как пользователь функции я хочу знать, почему отбрасывание возвращаемого значения – это проблема.

Оба этих аспекта были успешно решены в C++20. Конструкторы теперь могут тоже объявляться как `[[nodiscard]]`, и предупреждение может содержать дополнительную информацию.

Использование атрибута `[[nodiscard]]` в C++20

---

```
1  // nodiscardString.cpp
2
3  #include <utility>
4
5  struct MyType {
6
7      [[nodiscard("Implicit destroying of temporary MyInt.")]] MyType(int, bool) {}
8
9  };
10
11 template <typename T, typename ... Args>
12 [[nodiscard("You have a memory leak.")]]
13 T* create(Args&& ... args){
14     return new T(std::forward<Args>(args)...);
15 }
16
17 enum class [[nodiscard("Don't ignore the error code.")]] ErrorCode {
18     Okay,
19     Warning,
20     Critical,
21     Fatal
22 };
23
24 ErrorCode errorProneFunction() { return ErrorCode::Fatal; }
25
26 int main() {
27
28     int* val = create<int>(5);
29     delete val;
30
31     create<int>(5);
32
33     errorProneFunction();
34
35     MyType(5, true);
36
37 }
```

---

Теперь пользователь функций получает понятные сообщения. Ниже приводится вывод компилятора от Microsoft.

```

Windows PowerShell
C:\Users\rainer>cl.exe nodiscardString.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 19.27.29110 for x64
Copyright (C) Microsoft Corporation. All rights reserved.

nodiscardString.cpp
nodiscardString.cpp(31): warning C4858: discarding return value: You have a memory leak.
nodiscardString.cpp(33): warning C4858: discarding return value: Don't ignore the error code.
nodiscardString.cpp(35): warning C4858: discarding return value: Implicit destroying of temporary MyInt.
Microsoft (R) Incremental Linker Version 14.27.29110.0
Copyright (C) Microsoft Corporation. All rights reserved.

/out:nodiscardString.exe
nodiscardString.obj
C:\Users\rainer>

```

Компилятор C++20 выдает предупреждения об отброшенных объектах и кодах ошибки



### Проблема с `std::async`

Много существующих функций на C++ могут получить пользу от использования атрибута `[[nodiscard]]`. Идеальным кандидатом является функция `std::async`. Когда вы не используете значение, возвращенное `std::async`, тогда то, что предполагалось как асинхронный вызов, неявно становится синхронным. То, что должно было выполняться в отдельном потоке, вместо этого становится блокирующей функцией. Вы можете дополнительно прочитать про это крайне неочевидное поведение `std::async` в моем посте «The special Futures»<sup>1</sup>. Изучая синтаксис `[[nodiscard]]` на [cppreference.com/nodiscard](http://cppreference.com/nodiscard)<sup>2</sup>, я заметил, что объявление `std::async` в C++20 изменилось. Новое объявление приводится ниже:

`std::async` использует атрибут `[[nodiscard]]` в C++20

---

```

template<class Function, class... Args>
[[nodiscard]]
std::future<std::invoke_result_t<std::decay_t<Function>,
                                std::decay_t<Args>...>>
    async( Function&& f, Args&&... args );

```

---

Тип возвращаемого значения для `std::async` теперь объявлен как `[[nodiscard]]`.

Следующие два атрибута `[[likely]]` и `[[unlikely]]` относятся к оптимизации.

## 4.8.2 `[[likely]]` и `[[unlikely]]`

Предложение P0479R5<sup>3</sup> по атрибутам `[[likely]]` и `[[unlikely]]` является самым коротким предложением, которое я знаю. Чтобы дать вам представление об этом, вот интересное пояснение к нему: «Цель атрибута `[[likely]]` – позволить реализациям оптимизировать код для случая, когда какая-то часть кода явля-

<sup>1</sup> <https://www.modernescpp.com/index.php/the-special-futures>.

<sup>2</sup> <https://en.cppreference.com/w/cpp/language/attributes/nodiscard>.

<sup>3</sup> <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0479r5.html>.

ется более вероятным путем выполнения, чем другая часть, не имеющая подобного атрибута. Цель атрибута `[[unlikely]]` заключается в том, чтобы позволить реализациям оптимизировать код, когда какая-то ветвь является менее вероятной, чем другая. Ветвь выполнения включает метку тогда и только тогда, когда в ней содержится переход на эту метку. Чрезмерное использование каждого из этих атрибутов может привести к потере производительности».

Таким образом, оба этих атрибута предназначены для того, чтобы дать оптимизатору подсказки о более или менее вероятном пути выполнения.

Даем подсказку оптимизатору при помощи `[[likely]]`

---

```
for(size_t i=0; i < v.size(); ++i){
    if (v[i] < 0) [[likely]] sum -= sqrt(-v[i]);
    else sum += sqrt(v[i]);
}
```

---

Вопрос оптимизации продолжается атрибутом `[[no_unique_address]]`. На этот раз оптимизация производится с адресным пространством, а не с временем выполнения.

### 4.8.3 `[[no_unique_address]]`

Атрибут `[[no_unique_address]]` сообщает о том, что данный член класса не должен иметь адрес, отличный от остальных нестатических членов этого класса. Соответственно, если данный член класса является пустым, то компилятор может его оптимизировать так, чтобы он не занимал память.

Следующая программа показывает использование данного атрибута.

Использование атрибута `[[no_unique_address]]`

---

```
1 // uniqueAddress.cpp
2
3 #include <iostream>
4
5 struct Empty {};
6
7 struct NoUniqueAddress {
8     int d{};
9     [[no_unique_address]] Empty e{};
10 };
11
12 struct UniqueAddress {
13     int d{};
14     Empty e{};
15 };
16
```



---

```

17 int main() {
18
19     std::cout << '\n';
20
21     std::cout << std::boolalpha;
22
23     std::cout << "sizeof(int) == sizeof(NoUniqueAddress): "
24                 << (sizeof(int) == sizeof(NoUniqueAddress)) << '\n';
25
26     std::cout << "sizeof(int) == sizeof(UniqueAddress): "
27                 << (sizeof(int) == sizeof(UniqueAddress)) << '\n';
28
29     std::cout << '\n';
30
31     NoUniqueAddress NoUnique;
32
33     std::cout << "&NoUnique.d: " << &NoUnique.d << '\n';
34     std::cout << "&NoUnique.e: " << &NoUnique.e << '\n';
35
36     std::cout << '\n';
37
38     UniqueAddress unique;
39
40     std::cout << "&unique.d: " << &unique.d << '\n';
41     std::cout << "&unique.e: " << &unique.e << '\n';
42
43     std::cout << '\n';
44
45 }

```

---

Класс `NoUniqueAddress`, в отличие от класса `UniqueAddress` (строка 12), имеет размер, равный размеру типа `int` (строка 7). Члены `d` и `e` класса `UniqueAddress` (строки 40 и 41) имеют различные адреса, в отличие от членов класса `NoUniqueAddress` (строки 33 и 34).

```
1 // uniqueAddress.cpp
2
3 #include <iostream>
4
5 struct Empty {};
6
7 struct NoUniqueAddress {
8     int d{};
9     [[no_unique_address]] Empty e{};
10 };
11
12 struct UniqueAddress {
13     int d{};
14     Empty e{};
15 };
16
```

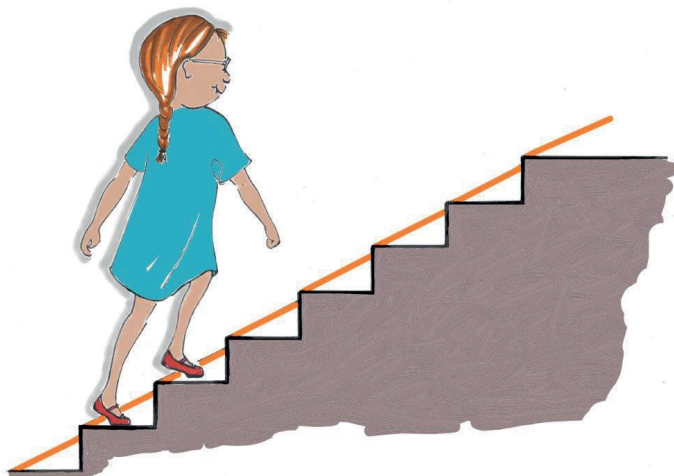
Использование классов NoUniqueAddress и UniqueAddress



### Важные замечания

- ♦ C++20 поддерживает несколько новых атрибутов. Атрибут `[[nodiscard("reason")]]` может быть использован в различных контекстах, чтобы проверить, не игнорируется ли возвращаемое функцией значение.
- ♦ Атрибуты `[[likely]]` и `[[unlikely]]` позволяют дать компилятору подсказку о том, какая ветвь кода более вероятна.
- ♦ Благодаря атрибуту `[[no_unique_address]]` члены класса могут иметь один и тот же адрес в памяти.

## 4.9 Дополнительные улучшения



Сиппи подымается наверх

В этом разделе описываются оставшиеся небольшие улучшения ядра языка в стандарте C++20.

### 4.9.1 `volatile`

Краткое описание предложения P1152R0<sup>1</sup> дает информацию об изменениях, которые претерпевает описатель `volatile`: «Предлагаемое устаревание (deprecation) сохраняет полезные применения `volatile` и удаляет сомнительные / уже не работающие. Цель состоит в том, чтобы изменить код времени компиляции, который сейчас работает не совсем корректно во время выполнения или через обновления с помощью компилятора».

Прежде чем я перейду непосредственно к `volatile`, я хочу ответить на принципиальный вопрос: когда вам следует использовать `volatile`? Замечание из стандарта C++ говорит, что «`volatile` – это подсказка компилятору не использовать агрессивную оптимизацию, включающую объект, поскольку объект может быть изменен путями, которые не были определены реализацией». Это значит, что для случая однопоточного выполнения компилятор должен выполнять операции загрузки и записи сразу, как только они встречаются в исходном коде. `Volatile`-операции не могут быть убраны или переупорядочены. Соответственно, вы можете использовать `volatile`-объекты для взаимодействия с обработчиком сигналов, но не с другими потоками.

Перед тем как я покажу вам, какая часть семантики `volatile` была сохранена, я хотел бы начать с того, что было убрано:

- объявлен устаревшим составной оператор присваивания `volatile` и преинкремент/постинкремент/декремент;

<sup>1</sup> <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1152r0.html>.

- объявлен устаревшим спецификатор `volatile` для параметров функции и возвращаемых типов;
- объявлен устаревшим спецификатор `volatile` в объявлении структурного связывания.

Если вы хотите узнать все детали, то я настоятельно советую вам посмотреть видео с CppCon 2019 под названием «Deprecating volatile»<sup>1</sup> от Дж. Ф. Бастиена. Вот несколько примеров из его выступления. Я исправил несколько опечаток в исходном коде. Числа в следующем фрагменте кода относятся к трем типам устаревания, перечисленным выше.

Устаревшие случаи использования `volatile`

---

```
// (1)
int neck, tail;
volatile int brachiosaur;
brachiosaur = neck;    // OK, a volatile store
tail = brachiosaur;    // OK, a volatile load

// deprecated: does this access brachiosaur once or twice
tail = brachiosaur = neck;

// deprecated: does this access brachiosaur once or twice
brachiosaur += neck;

// OK, a volatile load, an addition, a volatile store
brachiosaur = brachiosaur + neck;

#####
// (2)
// deprecated: a volatile return type has no meaning
volatile struct amber jurassic();

// deprecated: volatile parameters aren't meaningful to the
// caller, volatile only applies within the function
void trex(volatile short left_arm, volatile short right_arm);

// OK, the pointer isn't volatile, the data it points to is
void fly(volatile struct pterosaur* pterandon);
```

---

<sup>1</sup> [https://www.youtube.com/watch?v=KJW\\_DLaVXIY](https://www.youtube.com/watch?v=KJW_DLaVXIY).

```
#####
(3)
struct linhenykus { volatile short forelimb; };
void park(linhenykus alvarezsauroid) {
    // deprecated: does the binding copy the forelimbs?
    auto [what_is_this] = alvarezsauroid; // structured binding
    // ...
}
```



### **volatile и многопоточная семантика**

Обычно `volatile` используется, чтобы обозначить объекты, которые могут неожиданно измениться. Например, это могут быть объекты при программировании встраиваемых устройств, в которых реализован отображенный в память ввод/вывод (memory mapped I/O). Поскольку эти объекты могут изменяться независимо от программы и их значения записываются непосредственно в память, нет нужды в их оптимизированном хранении. Другими словами, `volatile` исключает агрессивную оптимизацию и не имеет никакого отношения к многопоточной семантике.

## 4.9.2 Оператор цикла `for` с инициализацией на основе диапазона

В C++20 вы можете использовать оператор цикла `for` с инициализацией на основе диапазона (range-based `for`).


Оператор цикла `for` с инициализацией на основе диапазона

```
1 // rangeBasedForLoopInitializer.cpp
2
3 #include <iostream>
4 #include <string>
5 #include <vector>
6
7 int main() {
8
9     for (auto vec = std::vector{1, 2, 3}; auto v : vec) {
10         std::cout << v << " ";
11     }
12
13     std::cout << "\n\n";
14 }
```

```
15     for (auto initList = {1, 2, 3}; auto e : initList) {
16         e *= e;
17         std::cout << e << " ";
18     }
19
20     std::cout << "\n\n";
21
22     using namespace std::string_literals;
23     for (auto str = "Hello World"s; auto c: str) {
24         std::cout << c << " ";
25     }
26
27     std::cout << '\n';
28
29 }
```

---

Оператор цикла `for` с инициализацией на основе диапазона используется в строке 9 для итерирования по `std::vector`, в строке 15 для итерирования по `std::initializer_list` и в строке 23 для итерирования по `std::string`. Более того, в строках 9 и 15 использован автоматический вывод типов для шаблонных классов, который появился в C++17. Вместо `std::vector<int>` я написал `std::vector`.



```
1 2 3
1 4 9
H e l l o   W o r l d
```

Использование оператора цикла `for` с инициализацией на основе диапазона

### 4.9.3 Виртуальная функция с `constexpr`

Функция с описателем `constexpr` может выполняться во время компиляции, но при этом также может выполняться и во время выполнения программы. Поэтому в C++20 вы можете делать их виртуальными. Виртуальная `constexpr`-функция может переопределить (override) не `constexpr`-функцию, и виртуальная не `constexpr`-функция может переопределить виртуальную `constexpr`-функцию. Я хочу подчеркнуть, что переопределение подразумевает, что соответствующая функция в базовом классе является виртуальной.

Программа `virtualConstexpr.cpp` демонстрирует оба этих варианта.

## Виртуальные constexpr-функции

---

```

1  // virtualConstexpr.cpp
2
3  #include <iostream>
4
5  struct X1 {
6      virtual int f() const = 0;
7  };
8
9  struct X2: public X1 {
10     constexpr int f() const override { return 2; }
11 };
12
13 struct X3: public X2 {
14     int f() const override { return 3; }
15 };
16
17 struct X4: public X3 {
18     constexpr int f() const override { return 4; }
19 };
20
21 int main() {
22
23     X1* x1 = new X4;
24     std::cout << "x1->f(): " << x1->f() << '\n';
25
26     X4 x4;
27     X1& x2 = x4;
28     std::cout << "x2.f(): " << x2.f() << '\n';
29
30 }
```

---

В строке 24 используется позднее связывание через указатель, строка 28 использует позднее связывание через ссылку.

```

1  // virtualConstexpr.cpp
2
3  #include <iostream>
4
5  struct X1 {
6      virtual int f() const = 0;
7  };
8
9  struct X2: public X1 {
10     constexpr int f() const override { return 2; }
11 };
12
13 struct X3: public X2 {
14     int f() const override { return 3; }
15 };
16
17 struct X4: public X3 {
18     constexpr int f() const override { return 4; }
19 };

```

Использование виртуальных constexpr-функций

#### 4.9.4 Новый символьный тип для utf8-строк: `char8_t`

В дополнение к символьным типам `char16_t` и `char32_t` из C++11 в C++20 добавлен еще один тип символов `char8_t`. Тип `char8_t` достаточно велик, чтобы вместить любой байт из UTF-8 (8 бит). У него тот же размер, знаковая и выравнивание, что и у `unsigned char`, но это отдельный тип.



##### **char против char8\_t**

Тип `char` – это один байт. В отличие от него, для `char8_t` число битов в байте не определено. Почти все реализации используют 8-битовый байт. Тип `std::string` – это обозначение для типа `std::basic_string` из `char`.

`std::string` и литерал `std::string`

---

```
std::string std::basic_string<char>
"Hello World"s
```

---

Соответственно, в C++20 был определен новый тип для `char8_t` (строка 1) и новый строковый литерал (строка 2).

Новый тип символов и строковый литерал UTF-8

---

```
1 std::u8string std::basic_string<char8_t>
2 u8"Hello World"
```

---

Программа `char8Str.cpp` демонстрирует использование нового типа `char8_t`.

Интуитивное использование нового символьного типа `char8_t`

---

```
1 // char8Str.cpp
2
3 #include <iostream>
4 #include <string>
5
6 int main() {
7
8     const char8_t* char8Str = u8"Hello world";
9     std::basic_string<char8_t> char8String = u8"helloWorld";
10    std::u8string char8String2 = u8"helloWorld";
11
12    char8String2 += u8".";
13
14    std::cout << "char8String.size(): " << char8String.size() << '\n';
15    std::cout << "char8String2.size(): " << char8String2.size() << '\n';
16
17    char8String2.replace(0, 5, u8"Hello ");
18
```



---

```

19  std::cout << "char8String2.size(): " << char8String2.size() << '\n';
20
21  }

```

---

Ниже приводится вывод этой программы.

---

```

1  std::ubstring std::basic_string<char8_t>
2  ub"Hello World"

```

---

Использование нового символьного типа `char8_t`

## 4.9.5 Использование `using enum` в локальной области видимости

Объявление `using enum` вводит объявленные в перечислении значения в текущую локальную область видимости.

Использование перечислений в локальной области видимости

---

```

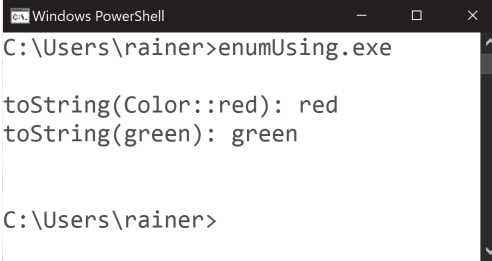
1  // enumUsing.cpp
2
3  #include <iostream>
4  #include <string_view>
5
6  enum class Color {
7      red,
8      green,
9      blue
10 };
11
12 std::string_view toString(Color col) {
13     switch (col) {
14         using enum Color;
15         case red:    return "red";
16         case green:  return "green";
17         case blue:   return "blue";
18     }
19     return "unknown";
20 }
21
22 int main() {
23

```

```
24  std::cout << '\n';
25
26  std::cout << "toString(Color::red): " << toString(Color::red) << '\n';
27
28  using enum Color;
29
30  std::cout << "toString(green): " << toString(green) << '\n';
31
32  std::cout << '\n';
33
34 }
```

---

Объявление `using enum` (строка 14) добавляет перечисления из `Color` в текущую локальную область видимости. Начиная с этого момента не нужно указывать `Color::` при обращении к ним (строки 15–17)<sup>1</sup>.



```
Windows PowerShell
C:\Users\rainer>enumUsing.exe

toString(Color::red): red
toString(green): green

C:\Users\rainer>
```

Применение `using enum`

## 4.9.6 Инициализаторы по умолчанию для битовых полей

Прежде всего разберемся, что такое битовое поле. Возьмем определение из Википедии: «Битовое поле – это структура данных, используемая в программировании компьютеров. Она состоит из набора последовательных мест в памяти, выделенных, чтобы хранить последовательность битов так, чтобы можно было обратиться к биту отдельно. Битовые поля чаще всего используются для представления целочисленных типов с фиксированным размером в битах».

В C++20 мы можем задать для битового поля начальное значение по умолчанию.

Инициализация по умолчанию для членов битового поля

---

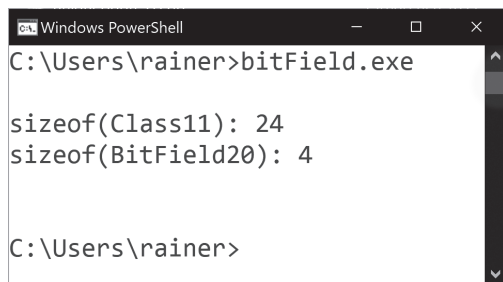
```
1  // bitField.cpp
2
3  #include <iostream>
4
```

---

<sup>1</sup> [https://en.wikipedia.org/wiki/Bit\\_field](https://en.wikipedia.org/wiki/Bit_field).

```
5 struct Class11 {
6     int i = 1;
7     int j = 2;
8     int k = 3;
9     int l = 4;
10    int m = 5;
11    int n = 6;
12 };
13
14 struct BitField20 {
15     int i : 3 = 1;
16     int j : 4 = 2;
17     int k : 5 = 3;
18     int l : 6 = 4;
19     int m : 7 = 5;
20     int n : 7 = 6;
21 };
22
23 int main () {
24
25     std::cout << '\n';
26
27     std::cout << "sizeof(Class11): " << sizeof(Class11) << '\n';
28     std::cout << "sizeof(BitField20): " << sizeof(BitField20) << '\n';
29
30     std::cout << '\n';
31
32 }
```

Так же, как и члены класса (строки 6–11) в C++11, в C++20 члены битового поля могут иметь инициализирующие значения по умолчанию (строки 15–20). Когда вы складываете числа 3, 4, 5, 6, 7 с числом 7, вы получите 32. Поэтому 32 бита, или ровно 4 байта, – это размер BitField20.



```
Windows PowerShell
C:\Users\rainer>bitField.exe

sizeof(Class11): 24
sizeof(BitField20): 4

C:\Users\rainer>
```

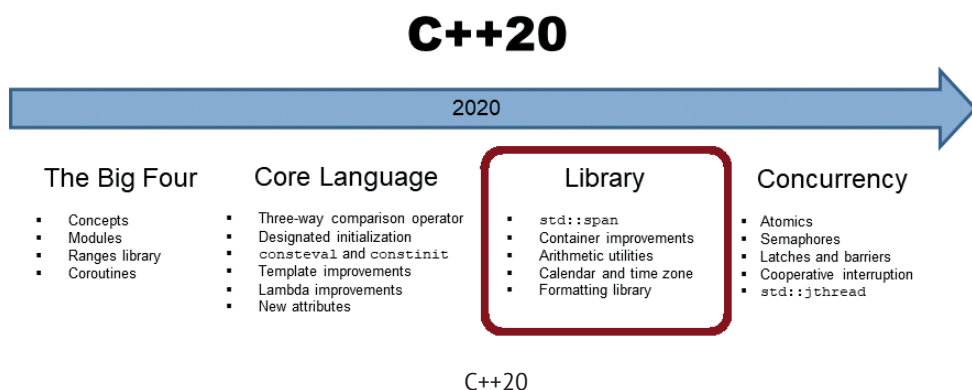
Информация о размере битового поля



### Важные замечания

- ♦ В C++20 было уточнено значение `volatile`. У `volatile` нет никакой многопоточной семантики, и он может быть использован только для предотвращения агрессивной оптимизации, когда объект может быть изменен независимо от логики течения программы.
- ♦ Оператор `for` с итерацией по диапазону значений может использовать инициализатор.
- ♦ Новый символьный тип `char8_t` достаточен для представления 8 бит информации.
- ♦ Директива `using enum` вводит перечисления в текущую локальную область видимости.
- ♦ У членов битового поля могут быть инициализаторы по умолчанию.
- ♦ Функции `constexpr` могут быть виртуальными.

## 5. Стандартная библиотека



Кроме библиотеки диапазонов, в стандартной библиотеке для стандарта C++20 есть много интересных возможностей, таких как `std::span` как нетипизированная ссылка на непрерывную область памяти, улучшенные реализации строк и контейнеров, улучшенные алгоритмы. Кроме того, к библиотеке `chrono` из C++11 была добавлена поддержка календарных и временных зон. Последнее, но при этом не менее важное: текст теперь может быть быстро и безопасно отформатирован.

## 5.1 Библиотека диапазонов



Сиппи рисует конвейер

Благодаря библиотеке диапазонов в стандарте C++20 работа со стандартной библиотекой шаблонов (STL) стала гораздо более комфортной и мощной. Алгоритмы библиотеки диапазонов являются «ленивыми» (lazy), т. е. могут работать непосредственно с контейнерами и совмещаться друг с другом. Комфорт и могущество библиотеки диапазонов обеспечиваются функциональными идеями, заложенными в основу библиотеки.

Прежде чем погружаться в детали, вот краткий пример использования библиотеки диапазонов.

Совмещение функций transform и filter

---

```
// rangesFilterTransform.cpp
```

```
#include <iostream>
#include <ranges>
#include <vector>

int main() {

    std::vector<int> numbers = {1, 2, 3, 4, 5, 6};

    auto results = numbers | std::views::filter([](int n){ return n % 2 == 0; })
                        | std::views::transform([](int n){ return n * 2; });

    for (auto v: results) std::cout << v << " ";    // 4 8 12
}
```

---

Вам нужно прочесть выражение слева направо. Символ `|` соответствует композиции функций: сначала будут отобраны только четные числа (`std::views::filter([](int n){ return n % 2 == 0; })`), после чего каждое из них будет удвоено (`std::views::transform([](int n){ return n * 2; })`). Этот небольшой пример показывает две новые «фишки» библиотеки диапазонов: композиция функций, которая к тому же может быть применена ко всему контейнеру.

Теперь вы готовы к деталям. Давайте начнем с того, что диапазоны и виды – это концепты.

### 5.1.1 Концепты `ranges` и `views`

Я уже показывал концепты `ranges` и `views` в главе, посвященной концептам. Поэтому приведу только краткую информацию:

- `range` – это группа элементов, по которой можно выполнять итерирование. Она предоставляет итератор `begin()` и маркер конца `end()`. Конечно, все контейнеры STL являются диапазонами.

Вид (`view`) – это то, что вы применяете к диапазону и выполняете какую-то операцию. Вид не владеет данными, и его временная сложность копирования, переноса и присваивания постоянна.

Виды, выполняемые над диапазоном

---

```
std::vector<int> numbers = {1, 2, 3, 4, 5, 6};
```

```
auto results = numbers | std::views::filter([](int n){ return n % 2 == 0; })
                      | std::views::transform([](int n){ return n * 2; });
```

---

В этом примере кода `numbers` – это диапазон, а `std::views::filter` и `std::views::transform` – это виды.

Благодаря видам с помощью C++20 можно выполнять разработку программ в функциональном стиле программирования. Виды можно комбинировать, и они «ленивые» (*lazy*). Я уже показал два вида, но в C++20 их гораздо больше.

| Вид                                                                              | Описание                                                                   |
|----------------------------------------------------------------------------------|----------------------------------------------------------------------------|
| <code>std::views::all_t</code><br><code>std::views::all</code>                   | Переводит диапазон в вид                                                   |
| <code>std::ranges::ref_view</code>                                               | Берет все элементы из другого диапазона                                    |
| <code>std::ranges::filter_view</code><br><code>std::views::filter</code>         | Выбирает элементы, удовлетворяющие предикату                               |
| <code>std::ranges::transform_view</code><br><code>std::views::transform</code>   | Преобразует каждый элемент                                                 |
| <code>std::ranges::take_view</code><br><code>std::views::take</code>             | Берет первые <code>n</code> элементов другого вида                         |
| <code>std::ranges::take_while_view</code><br><code>std::views::take_while</code> | Берет элементы из другого вида, пока предикат возвращает <code>true</code> |

| Вид                                                                                    | Описание                                                                               |
|----------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| <code>std::ranges::drop_view</code><br><code>std::views::drop</code>                   | Пропускает первые <i>n</i> элементов из другого вида                                   |
| <code>std::ranges::drop_while_view</code><br><code>std::views::drop_while</code>       | Пропускает первые элементы из другого вида, пока предикат не вернет <code>false</code> |
| <code>std::ranges::join_view</code><br><code>std::views::join</code>                   | Соединяет вид диапазонов                                                               |
| <code>std::ranges::split_view</code><br><code>std::views::split</code>                 | Разделяет вид, используя ограничитель                                                  |
| <code>std::ranges::common_view</code><br><code>std::views::common</code>               | Переводит вид в <code>std::ranges::common_range</code>                                 |
| <code>std::ranges::reverse_view</code><br><code>std::views::reverse</code>             | Производит обход в обратном порядке                                                    |
| <code>std::ranges::basic_istream_view</code><br><code>std::ranges::istream_view</code> | Применяет <code>operator&gt;&gt;</code> к входному потоку                              |
| <code>std::ranges::elements_view</code><br><code>std::views::elements</code>           | Создает вид из <i>n</i> -го элемента кортежа                                           |
| <code>std::ranges::keys_view</code><br><code>std::views::keys</code>                   | Создает вид из первого элемента значений типа пары                                     |
| <code>std::ranges::values_view</code><br><code>std::views::values</code>               | Создает вид из второго элемента значений типа пары                                     |

В общем случае вы можете использовать вид вроде `std::views::transform` с альтернативным именем `std::views::transform_view`.

## 5.1.2 Работа алгоритмов непосредственно со всем контейнером

Алгоритмы из стандартной библиотеки шаблонов (STL) иногда бывают неудобными. Им нужно сразу два итератора – начала и конца. И это часто больше, чем то, что вы хотели бы написать.

Алгоритмы из STL требуют сразу двух итераторов – начала и конца

---

```
// sortClassical.cpp
```

```
#include <algorithm>
#include <iostream>
#include <vector>
```



---

```
int main() {

    std::vector<int> myVec{-3, 5, 0, 7, -4};
    std::sort(myVec.begin(), myVec.end());
    for (auto v: myVec) std::cout << v << " "; // -4, -3, 0, 5, 7

}
```

---

Не было бы здорово, если бы `std::sort` мог быть выполнен над всем контейнером? Благодаря библиотеке диапазонов в C++20 это возможно.

Алгоритмы из библиотеки диапазонов работают непосредственно со всем контейнером

---

*// sortRanges.cpp*

```
#include <algorithm>
#include <iostream>
#include <vector>

int main() {

    std::vector<int> myVec{-3, 5, 0, 7, -4};
    std::ranges::sort(myVec);
    for (auto v: myVec) std::cout << v << " "; // -4, -3, 0, 5, 7

}
```

---

Алгоритмы из библиотеки алгоритмов<sup>1</sup>, содержащиеся в заголовочном файле `<algorithm>`<sup>2</sup>, такие как `std::sort`, имеют специализированные версии, такие как `std::ranges::sort`.

Когда вы изучите перегрузки `std::ranges::sort`, то вы увидите, что они поддерживают проецирование (projection) алгоритма на весь контейнер.

### 5.1.2.1 Проекция алгоритма на весь контейнер

У `std::ranges::sort` есть два перегруженных варианта.

Перегрузки `std::ranges::sort`

---

```
template< std::random_access_iterator I, std::sentinel_for<I> S,
          class Comp = ranges::less, class Proj = std::identity >
requires std::sortable<I, Comp, Proj>
constexpr I sort( I first, S last, Comp comp = {}, Proj proj = {} );
```

---

<sup>1</sup> <https://en.cppreference.com/w/cpp/algorithm>.

<sup>2</sup> <https://en.cppreference.com/w/cpp/header/algorithm>.

```
template< ranges::random_access_range R, class Comp = ranges::less,
        class Proj = std::identity >
requires std::sortable<ranges::iterator_t<R>, Comp, Proj>
constexpr ranges::borrowed_iterator_t<R> sort( R&&r, Comp comp = {},
        Proj proj = {} \ );
```

---

Когда вы посмотрите на второй перегруженный вариант, то заметите, что он принимает на вход сортируемый диапазон `R`, предикат `Comp` и проекцию `Proj`. Предикат `Comp` по умолчанию равен `less`, а проекция `Proj` по умолчанию равна тождественному преобразованию.

Проекция – это отображение множества на подмножество. Давайте разберемся, что это значит.

Применение проекции к типам данных

---

```
// rangeProjection.cpp
```

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <vector>

struct PhoneBookEntry{
    std::string name;
    int number;
};

void printPhoneBook(const std::vector<PhoneBookEntry>& phoneBook) {
    for (const auto& entry: phoneBook) std::cout << "(" << entry.name << ", "
        << entry.number << ")";

    std::cout << "\n\n";
}

int main() {

    std::cout << '\n';

    std::vector<PhoneBookEntry> phoneBook{ {"Brown", 111}, {"Smith", 444},
        {"Grimm", 666}, {"Butcher", 222}, {"Taylor", 555}, {"Wilson", 333} };

    std::ranges::sort(phoneBook, {}, &PhoneBookEntry::name); //a scending
    printPhoneBook(phoneBook);                                //by name

    std::ranges::sort(phoneBook, std::ranges::greater() ,
```

---

```

std::ranges::sort(phoneBook, std::ranges::greater() ,
                  &PhoneBookEntry::name);           //a scending
printPhoneBook(phoneBook);                          //by name

std::ranges::sort(phoneBook, {}, &PhoneBookEntry::number); //a scending
printPhoneBook(phoneBook);                          //by name

std::ranges::sort(phoneBook, std::ranges::greater(),
                  &PhoneBookEntry::number);         //a scending
printPhoneBook(phoneBook);                          //by name

std::cout << '\n';

}

```

---

Массив `phoneBook` (строка 23) состоит из структур типа `PhoneBookEntry` (строка 8). `PhoneBookEntry` состоит из `name` и `number`. Благодаря проекции `phoneBook` может быть отсортирован по возрастанию по имени (строка 26), по убыванию по имени (строка 29), по возрастанию `number` (строка 33) и по убыванию `number` (строка 36).

```

(Brown, 111) (Butcher, 222) (Grimm, 666) (Smith, 444) (Taylor, 555) (Wilson, 333)
(Wilson, 333) (Taylor, 555) (Smith, 444) (Grimm, 666) (Butcher, 222) (Brown, 111)
(Brown, 111) (Butcher, 222) (Wilson, 333) (Smith, 444) (Taylor, 555) (Grimm, 666)
(Grimm, 666) (Taylor, 555) (Smith, 444) (Wilson, 333) (Butcher, 222) (Brown, 111)

```

#### Применение проекции к типам данных

Большинство алгоритмов над диапазонами поддерживают проекцию.

##### 5.1.2.2 Виды по ключам и значениям

Также вы можете создавать виды по ключам (строка 16) и значениям (строка 26) для `std::unordered_map`.

Виды по ключам и значениям для `std::unordered_map`

---

```

1 // rangesEntireContainer.cpp
2
3 #include <iostream>
4 #include <ranges>
5 #include <string>
6 #include <unordered_map>
7
8

```

```
9 int main() {
10
11     std::unordered_map<std::string, int> freqWord{ {"witch", 25}, {"wizard", 33},
12                                                    {"tale", 45}, {"dog", 4},
13                                                    {"cat", 34}, {"fish", 23} };
14
15     std::cout << "Keys: " << '\n';
16     auto names = std::views::keys(freqWord);
17     for (const auto& name : names){ std::cout << name << " "; }
18     std::cout << '\n';
19     for (const auto& name : std::views::keys(freqWord)){ std::cout << name << " "; }
20
21     std::cout << "\n\n";
22
23     std::cout << "Values: " << '\n';
24     auto values = std::views::values(freqWord);
25     for (const auto& value : values){ std::cout << value << " "; }
26     std::cout << '\n';
27     for (const auto& value : std::views::values(freqWord)) {
28         std::cout << value << " ";
29     }
30
31 }
```

---

Следует отметить, что ключи и значения могут быть выведены и непосредственно (строки 19 и 27). Вывод будет идентичен.

---

```
1 // rangesEntireContainer.cpp
2
3 #include <iostream>
4 #include <ranges>
5 #include <string>
6 #include <unordered_map>
7
8
9 int main() {
10
11     std::unordered_map<std::string, int> freqWord{ {"witch", 25}, {"wizard", 33},
12                                                    {"tale", 45}, {"dog", 4},
13                                                    {"cat", 34}, {"fish", 23} };
14
15     std::cout << "Keys: " << '\n';
16     auto names = std::views::keys(freqWord);
17     for (const auto& name : names){ std::cout << name << " "; }
18     std::cout << '\n';
19     for (const auto& name : std::views::keys(freqWord)){ std::cout << name << " "; }
```

Виды ключей и значений для `std::unordered_map`

Работать непосредственно с контейнером может быть не так захватывающе, но вот композиция и отложенное выполнение (ленивое выполнение, *lazy evaluation*) являются захватывающими.

### 5.1.3 Композиция функций

В примере `rangesComposition.cpp` я использую `std::map`, поскольку упорядочение ключей является здесь существенным.

Композиция видов

---

```
1 // rangesComposition.cpp
2
```

---

```

3 #include <iostream>
4 #include <ranges>
5 #include <string>
6 #include <map>
7
8
9 int main() {
10
11     std::map<std::string, int> freqWord{ {"witch", 25}, {"wizard", 33},
12                                           {"tale", 45}, {"dog", 4},
13                                           {"cat", 34}, {"fish", 23} };
14
15     std::cout << "All words: ";
16     for (const auto& name : std::views::keys(freqWord)) { std::cout << name << " "; }
17
18     std::cout << '\n';
19
20     std::cout << "All words, reverses: ";
21     for (const auto& name : std::views::keys(freqWord)
22           | std::views::reverse) { std::cout << name << " "; }
23
24     std::cout << '\n';
25
26     std::cout << "The first 4 words: ";
27     for (const auto& name : std::views::keys(freqWord)
28           | std::views::take(4)) { std::cout << name << " "; }
29
30     std::cout << '\n';
31
32     std::cout << "All words starting with w: ";
33     auto firstw = [](const std::string& name){ return name[0] == 'w'; };
34     for (const auto& name : std::views::keys(freqWord)
35           | std::views::filter(firstw)) { std::cout << name << " "; }
36
37     std::cout << '\n';
38
39 }

```

---

Меня интересуют только ключи. Я вывожу их все в обычном порядке (строка 15), все в обратном порядке (строка 20), первые четыре (строка 26) и ключи, начинающиеся с буквы ‘w’ (строка 32).

Вывод программы следующий:

---

```

1 // rangesComposition.cpp
2
3 #include <iostream>
4 #include <ranges>
5 #include <string>
6 #include <map>
7

```

Символ `|` – это синтаксический сахар (syntax sugar)<sup>1</sup> для композиции функций. Вместо записи `C(R)` вы можете записать `R | C`. Соответственно, следующие строки эквивалентны.

Три синтаксические формы композиции:

---

```
auto rev1 = std::views::reverse(std::views::keys(freqWord));
auto rev2 = std::views::keys(freqWord) | std::views::reverse;
auto rev3 = freqWord | std::views::keys | std::views::reverse;
```

---

### 5.1.4 Отложенное выполнение

`std::views::iota` – это фабрика диапазонов для создания последовательностей элементов путем последовательного увеличения начального значения. Эта последовательность может быть конечной или бесконечной. Программа `rangesIota.cpp` заполняет `std::vector` 10 целыми числами, начиная с 0.

Использование `std::views::iota` для заполнения `std::vector`

---

```
1 // rangesIota.cpp
2
3 #include <iostream>
4 #include <numeric>
5 #include <ranges>
6 #include <vector>
7
8 int main() {
9
10     std::cout << std::boolalpha;
11
12     std::vector<int> vec;
13     std::vector<int> vec2;
14
15     for (int i: std::views::iota(0, 10)) vec.push_back(i);
16
17     for (int i: std::views::iota(0) | std::views::take(10)) vec2.push_back(i);
18
19     std::cout << "vec == vec2: " << (vec == vec2) << '\n';
20
21     for (int i: vec) std::cout << i << " ";
22
23 }
```

---

Первый вызов `iota` (строка 15) создает все числа от 0 до 9 с шагом 1. Второй вызов `iota` (строка 17) создает бесконечный поток данных, начиная с 0 и с шагом 1. На самом деле `std::views::iota(0)` использует отложенные вычисления.

---

<sup>1</sup> [https://en.wikipedia.org/wiki/Syntactic\\_sugar](https://en.wikipedia.org/wiki/Syntactic_sugar).

Я получу значение, только если я попрошу об этом. Я прошу его 10 раз. Поэтому оба вектора одинаковы.

```
vec == vec2: true
0 1 2 3 4 5 6 7 8 9
```

Использование `std::views::iota` для заполнения `std::vector`

Теперь я хочу решить более сложную задачу: найти первые 20 простых чисел, начиная с 1 000 000.

Первые 20 простых чисел, начиная с 1 000 000

---

```
1 // rangesLazy.cpp
2
3 #include <iostream>
4 #include <ranges>
5
6
7 bool isPrime(int i) {
8     for (int j=2; j*j <= i; ++j){
9         if (i % j == 0) return false;
10    }
11    return true;
12 }
13
14 int main() {
15
16     std::cout << "Numbers from 1'000'000 to 1'001'000 (displayed each 100th): "
17               << '\n';
18     for (int i: std::views::iota(1'000'000, 1'001'000)) {
19         if (i % 100 == 0) std::cout << i << " ";
20     }
21
22     std::cout << "\n\n";
23
24     auto odd = [](int i){ return i % 2 == 1; };
25     std::cout << "Odd numbers from 1'000'000 to 1'001'000 (displayed each 100th): "
26               << '\n';
27     for (int i: std::views::iota(1'000'000, 1'001'000) | std::views::filter(odd)) {
28         if (i % 100 == 1) std::cout << i << " ";
29     }
30
31     std::cout << "\n\n";
32
33     std::cout << "Prime numbers from 1'000'000 to 1'001'000: " << '\n';
34     for (int i: std::views::iota(1'000'000, 1'001'000) | std::views::filter(odd)
35           | std::views::filter(isPrime)) {
36         std::cout << i << " ";
37     }
```

```
38
39  std::cout << "\n\n";
40
41  std::cout << "20 prime numbers starting with 1'000'000: " << '\n';
42  for (int i: std::views::iota(1'000'000) | std::views::filter(odd)
43      | std::views::filter(isPrime)
44      | std::views::take(20)) {
45      std::cout << i << " ";
46  }
47
48  std::cout << '\n';
49
50 }
```

---

Вот моя итеративная стратегия:

- строка 18: конечно, я не знаю, когда у меня будет более 20 простых чисел, больших 1 000 000. Для надежности я создаю 1000 чисел. По понятным причинам я показываю каждое 100-е значение;
- строка 27: меня интересуют только нечетные числа, поэтому я убираю все четные;
- строка 34: теперь время применить следующий фильтр. Предикат `isPrime` (строка 7) проверяет, является ли число простым. Как вы можете увидеть на следующем скриншоте, у меня набралось 75 простых чисел;
- строка 42: лень – это достоинство. Я использую `std::iota` как бесконечный генератор чисел, начиная с 1 000 000, и прошу ровно 20 простых чисел.

```
Numbers from 1'000'000 to 1'001'000 (displayed each 100th):
1000000 1000100 1000200 1000300 1000400 1000500 1000600 1000700 1000800 1000900

Odd numbers from 1'000'000 to 1'001'000 (displayed each 100th):
1000001 1000101 1000201 1000301 1000401 1000501 1000601 1000701 1000801 1000901

Prime numbers from 1'000'000 to 1'001'000:
1000003 1000033 1000037 1000039 1000081 1000099 1000117 1000121 1000133 1000151
1000159 1000171 1000183 1000187 1000193 1000199 1000211 1000213 1000231 1000249
1000253 1000273 1000289 1000291 1000303 1000313 1000333 1000357 1000367 1000381
1000393 1000397 1000403 1000409 1000423 1000427 1000429 1000453 1000457 1000507
1000537 1000541 1000547 1000577 1000579 1000589 1000609 1000619 1000621 1000639
1000651 1000667 1000669 1000679 1000691 1000697 1000721 1000723 1000763 1000777
1000793 1000829 1000847 1000849 1000859 1000861 1000889 1000907 1000919 1000921
1000931 1000969 1000973 1000981 1000999

20 prime numbers starting with 1'000'000:
1000003 1000033 1000037 1000039 1000081 1000099 1000117 1000121 1000133 1000151
1000159 1000171 1000183 1000187 1000193 1000199 1000211 1000213 1000231 1000249
```

Первые 20 простых чисел, начиная с 1 000 000



## 5.1.5 Определение видов

Вы можете определить свой собственный вид.

### 5.1.5.1 `std::ranges::view_interface`

Благодаря `std::ranges::view_interface`<sup>1</sup> определить вид очень просто. Для соответствия концепту вида вам необходимо, чтобы ваш вид содержал как минимум конструктор по умолчанию и методы `begin()` и `end()`:

Создание пользовательского вида

---

```
class MyView : public std::ranges::view_interface<MyView> {
public:
    auto begin() const { /*...*/ }
    auto end() const { /*...*/ }
};
```

---

Публично наследуя `MyView` от `std::ranges::view_interface`, используя себя в качестве параметра шаблона, `MyView` становится видом. Этот прием использования шаблонного класса, когда сам класс выступает в качестве параметра шаблона, называется рекурсивный шаблон (Curiously Recurring Template Pattern<sup>2</sup>, CRTP).

Я использую этот прием в следующем примере для создания вида из контейнера из стандартной библиотеки шаблонов.

### 5.1.5.2 Вид контейнера

Вид `ContainerView` создает вид для произвольного контейнера.

Создание вида по контейнеру

---

```
1 // containerView.cpp
2
3 #include <iostream>
4 #include <ranges>
5 #include <string>
6 #include <vector>
7
8 template<std::ranges::input_range Range>
9 requires std::ranges::view<Range>
10 class ContainerView : public std::ranges::view_interface<ContainerView<Range>> {
11 private:
12     Range range_{};
13     std::ranges::iterator_t<Range> begin_{ std::begin(range_) };
14     std::ranges::iterator_t<Range> end_{ std::end(range_) };
15 }
```

---

<sup>1</sup> [https://en.cppreference.com/w/cpp/ranges/view\\_interface](https://en.cppreference.com/w/cpp/ranges/view_interface).

<sup>2</sup> <https://www.modernescpp.com/index.php/c-is-still-lazy>.

```
16 public:
17     ContainerView() = default;
18
19     constexpr ContainerView(Range r): range_(std::move(r)) ,
20                                     begin_(std::begin(r)), end_(std::end(r)) {}
21
22     constexpr auto begin() const {
23         return begin_;
24     }
25     constexpr auto end() const {
26         return end_;
27     }
28 };
29
30 template<typename Range>
31 ContainerView(Range&& range) -> ContainerView<std::ranges::views::all_t<Range>>;
32
33 int main() {
34
35     std::vector<int> myVec{ 1, 2, 3, 4, 5, 6, 7, 8, 9};
36
37     auto myContainerView = ContainerView(myVec);
38     for (auto c : myContainerView) std::cout << c << " ";
39     std::cout << '\n';
40
41     for (auto i : std::views::reverse(ContainerView(myVec))) std::cout << i << ' ';
42     std::cout << '\n';
43
44     for (auto i : ContainerView(myVec) | std::views::reverse) std::cout << i << ' ';
45     std::cout << '\n';
46
47     std::cout << std::endl;
48
49     std::string myStr = "Only for testing purpose.";
50
51     auto myContainerView2 = ContainerView(myStr);
52     for (auto c: myContainerView2) std::cout << c << " ";
53     std::cout << '\n';
54
55     for (auto i : std::views::reverse(ContainerView(myStr))) std::cout << i << ' ';
56     std::cout << '\n';
57
58     for (auto i : ContainerView(myStr) | std::views::reverse) std::cout << i << ' ';
59     std::cout << '\n';
60
61 }
```

---

Шаблонный класс `ContainerView` (строка 8) наследуется от класса `std::ranges::view_interface` и требует, чтобы контейнер поддерживал концепт `std::ranges::view` (строка 9). Оставшаяся часть реализации очень простая.

У `ContainerView` есть конструктор по умолчанию (строка 17) и два требуемых метода – `begin ()` и `end ()` (строки 22 и 25). Для удобства добавил я пользовательскую подсказку к тому, как определить параметры шаблонного класса по его списку аргументов (class template argument deduction, строка 32).

В функции `main` я применяю `ContainerView` к `std::vector` (строка 37) и `std::string` (строка 49), а также выполняю итерирование по ним вперед и назад.

```
1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1
9 8 7 6 5 4 3 2 1

o n l y   f o r   t e s t i n g   p u r p o s e .
. e s o p r u p   g n i t s e t   r o f   y l n o
. e s o p r u p   g n i t s e t   r o f   y l n o
```

Создание вида из контейнера

Приведу несколько слов подсказки того, как определить параметры шаблонного класса по его списку аргументов.



### Class Template Argument Deduction Guide

Начиная со стандарта C++17 компилятор может определить параметры шаблона из его аргументов. Template Deduction Guide – это шаблон для компилятора, как выводить аргументы шаблона.

Когда вы используете `ContainerView(myVec)`, компилятор применяет следующую задаваемую пользователем подсказку.

Задаваемая пользователем подсказка по определению параметров шаблона

---

```
template<class Range>
```

```
ContainerView(Range&& range) -> ContainerView<std::ranges::views::all_t<Range>>;
```

---

По сути, вызов `ContainerView(myVec)` заставляет компилятор инстанцировать код справа от `->`.

Применение правила вывода для `ContainerView(myVec)`

---

```
ContainerView<std::ranges::views::all_t<std::vector<int>&>>>(myVec);
```

---

Сайт [cppreference.com](https://en.cppreference.com/w/cpp/language/class_template_argument_deduction)<sup>1</sup> содержит дополнительную информацию по задаваемой пользователем подсказке по определению параметров шаблона.

В следующем разделе про библиотеку диапазонов я хочу провести небольшой эксперимент. Смогу ли я приемы работы с кодом, используемые в Python, применить в C++?

<sup>1</sup> [https://en.cppreference.com/w/cpp/language/class\\_template\\_argument\\_deduction](https://en.cppreference.com/w/cpp/language/class_template_argument_deduction).

## 5.1.6 Аромат Python

В языке программирования Python<sup>1</sup> есть удобные функции `filter` и `map`.

- `filter`: применяет предикат ко всем элементам итерируемого объекта и возвращает те элементы, для которых предикат вернул `true`;
- `map`: применяет функцию ко всем элементам итерируемого объекта и возвращает новый итерируемый объект, состоящий из преобразованных элементов.

Итерируемым объектом в C++ может быть тип, который можно использовать в циклах, основанных на диапазоне.

Также Python позволяет комбинировать обе эти функции, получая объекты, принимающие на вход итерируемые объекты и возвращающие итерируемые объекты (`list comprehension`).

Вот что я хочу сделать: я хочу реализовать функции `filter` и `map`, которые можно комбинировать описанным образом в Python, но на языке C++ при помощи библиотеки диапазонов.

### 5.1.6.1 `filter`

Функция `filter` из Python может быть непосредственно отображена в соответствующую функцию из библиотеки диапазонов.

Функция `filter` из Python на C++

---

```
1  // filterRanges.cpp
2
3  #include <iostream>
4  #include <numeric>
5  #include <ranges>
6  #include <string>
7  #include <vector>
8
9  template <typename Func, typename Seq>
10 auto filter(Func func, const Seq& seq) {
11
12     typedef typename Seq::value_type value_type;
13
14     std::vector<value_type> result{};
15     for (auto i : seq | std::views::filter(func)) result.push_back(i);
16
17     return result;
18 }
19
20
```

---

<sup>1</sup> <https://www.python.org/>.

```

21 int main() {
22
23     std::cout << '\n';
24
25     std::vector<int> myInts(50);
26     std::iota(myInts.begin(), myInts.end(), 1);
27     auto res = filter([](int i){ return (i % 3) == 0; }, myInts);
28     for (auto v: res) std::cout << v << " ";
29
30
31     std::vector<std::string> myStrings{"Only", "for", "testing", "purposes"};
32     auto res2 = filter([](const std::string& s){ return std::isupper(s[0]); },
33                       myStrings);
34
35     std::cout << "\n\n";
36
37     for (auto word: res2) std::cout << word << '\n';
38
39     std::cout << '\n';
40
41 }

```

Прежде чем я продолжу говорить о программе, я хотел бы показать ее вывод.

```

3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48

Only

```

Результат применения функции filter

Функция filter (строка 9) довольно легко читаема. Строка 12 определяет тип элемента. Я просто применяю func к каждому элементу последовательности и возвращаю элементы в std::vector. Строка 27 выбирает все элементы от 1 до 50, для которых справедливо выражение  $(i \% 3) == 0$ . Только строки, начинающиеся с заглавной буквы, пройдут фильтр в строке 32.

### 5.1.6.2. map

map применяет вызываемый объект к каждому элементу входной последовательности.

## Реализация функции map из Python на C++

---

```
1  // mapRanges.cpp
2
3  #include <iostream>
4  #include <list>
5  #include <ranges>
6  #include <string>
7  #include <vector>
8  #include <utility>
9
10
11 template <typename Func, typename Seq>
12 auto map(Func func, const Seq& seq) {
13
14     typedef typename Seq::value_type value_type;
15     using return_type = decltype(func(std::declval<value_type>()));
16
17     std::vector<return_type> result{};
18     for (auto i : seq | std::views::transform(func)) result.push_back(i);
19
20     return result;
21 }
22
23 int main() {
24
25     std::cout << '\n';
26
27     std::list<int> myInts{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
28     auto res = map([](int i){ return i * i; }, myInts);
29
30     for (auto v: res) std::cout << v << " ";
31
32     std::cout << "\n\n";
33
34     std::vector<std::string> myStrings{"Only", "for", "testing", "purposes"};
35     auto res2 = map([](const std::string& s){ return std::make_pair(s.size(),
36                                                                    s); }, myStrings);
37
38     for (auto p: res2) std::cout << "(" << p.first << ", " << p.second << ")" << " ";
39
40     std::cout << "\n\n";
41
42 }
```

---

В определении функции `map` очень интересна строка 15. Выражение `decltype(func(std::declval<value_type>()))` выводит `return_type`. Это тот тип, к которому приводятся элементы входной последовательности, если мы применим к ним функцию `func`. Выражение `std::declval<value_type>()` возвращает `rvalue`-ссылку, по которой `decltype` может определить тип. Это значит, что вызов `map([](int i){ return i * i; }, myInts)` (строка 28) отображает каждый элемент `myInt` в его возведение в квадрат, а вызов `map([](const std::string& s) { return std::make_pair(s.size(), s); }, myStrings)` отображает каждую строку из `myString` в пару (`pair`). Первым элементом такой пары является длина строки.

```
1 4 9 16 25 36 49 64 81 100

(4, Only) (3, for) (7, testing) (8, purposes)
```

Применение функции `map`

### 5.1.6.3 Комбинация функций

Программа `listComprehensionRanges.cpp` является упрощенной версией соответствующего алгоритма на Python.

Функция `map` применяется к каждому элементу входной последовательности.

Упрощенный вариант комбинации функций

---

```
1 // listComprehensionRanges.cpp
2
3 #include <algorithm>
4 #include <cctype>
5 #include <functional>
6 #include <iostream>
7 #include <ranges>
8 #include <string>
9 #include <vector>
10 #include <utility>
11
12 template <typename T>
13 struct AlwaysTrue {
14     constexpr bool operator()(const T&) const {
15         return true;
16     }
17 };
18
19 template <typename Map, typename Seq, typename Filt = AlwaysTrue<
20                                     typename Seq::value_type>>
21 auto mapFilter(Map map, Seq seq, Filt filt = Filt()) {
22
23     typedef typename Seq::value_type value_type;
24     using return_type = decltype(map(std::declval<value_type>()));
```

```
25
26     std::vector<return_type> result{};
27     for (auto i : seq | std::views::filter(filt)
28         | std::views::transform(map)) result.push_back(i);
29     return result;
30 }
31
32 int main() {
33
34     std::cout << '\n';
35
36     std::vector myInts{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
37
38     auto res = mapFilter([](int i){ return i * i; }, myInts);
39     for (auto v: res) std::cout << v << " ";
40
41     std::cout << "\n\n";
42
43     res = mapFilter([](int i){ return i * i; }, myInts,
44                   [](auto i){ return i % 2 == 1; });
45     for (auto v: res) std::cout << v << " ";
46
47     std::cout << "\n\n";
48
49     std::vector<std::string> myStrings{"Only", "for", "testing", "purposes"};
50     auto res2 = mapFilter([](const std::string& s){
51         return std::make_pair(s.size(), s);
52     }, myStrings);
53     for (auto p: res2) std::cout << "(" << p.first << ", " << p.second << ") " ;
54
55     std::cout << "\n\n";
56
57     myStrings = {"Only", "for", "testing", "purposes"};
58     res2 = mapFilter([](const std::string& s){
59         return std::make_pair(s.size(), s);
60     }, myStrings,
61                   [](const std::string& word){ return std::isupper(word[0]); });
62
63     for (auto p: res2) std::cout << "(" << p.first << ", " << p.second << ") " ;
64
65     std::cout << "\n\n";
66
67 }
```

---

Предикат по умолчанию, который функция `filter` использует, всегда возвращает `true` (строка 12). Это значит, что по умолчанию функция `mapFilter`



всегда ведет себя как функция `map`. Именно так себя ведет функция `mapFilter` в строках 37 и 49. В строках 49 и 55 в одном вызове сразу применяются функции `map` и `filter`.

```
1 4 9 16 25 36 49 64 81 100

1 9 25 49 81

(4, Only) (3, for) (7, testing) (8, purposes)

(4, Only)
```

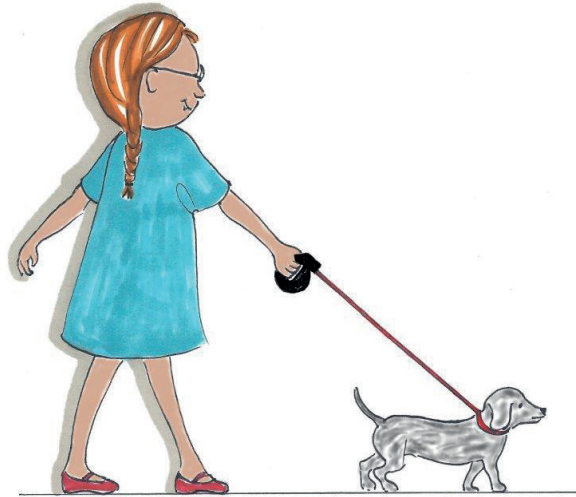
Применение обеих функций `map` и `filter`



### Краткая информация

- ♦ Библиотека диапазонов предоставляет дополнительные версии алгоритмов из STL. Алгоритмы в библиотеке диапазонов являются отложенными (*lazy*). Это значит, что они могут применяться непосредственно к контейнерам и могут комбинироваться друг с другом.
- ♦ Алгоритмы из библиотеки диапазонов:
  - являются отложенными и поэтому могут применяться к бесконечным потокам данных;
  - могут работать непосредственно с контейнером, и им не нужен диапазон, задаваемый парой итераторов;
  - могут компоноваться друг с другом при помощи символа `|`.

## 5.2 std::span



Сиппи гуляет с собакой

`std::span` – это объект, который ссылается на непрерывную последовательность объектов. Иногда `std::span` называют видом, и при этом он никогда не является владельцем данных. Непрерывная последовательность объектов может быть массивом языка C, указателем с размером, `std::array`, `std::vector` или `std::string`.

У `std::span` может быть статическая (*static extent*) или динамическая длина (изменяемая, *dynamic extent*). По умолчанию у `std::span` динамическая длина.

Определение `std::span`

---

```
template <typename T, std::size_t Extent = std::dynamic_extent>
class span;
```

---

### 5.2.1 Статическая и динамическая длина

Когда у `std::span` статическая длина, то его размер известен во время компиляции и является частью типа: `std::span<T, size>`. Соответственно, реализации нужен только указатель на первый элемент из непрерывной последовательности объектов.

Реализация `std::span` с динамической длиной состоит из указателя на первый элемент и размера непрерывной последовательности объектов. Размер не является частью типа: `std::span<T>`.

Следующий пример `staticDynamicExtentSpan.cpp` выделяет различия между обоими этими случаями.

---

`std::span` со статической и динамической длиной

---

```

1  // staticDynamicExtentSpan.cpp
2
3  #include <iostream>
4  #include <span>
5  #include <vector>
6
7  void printMe(std::span<int> container) {
8
9      std::cout << "container.size(): " << container.size() << '\n';
10     for (auto e : container) std::cout << e << ' ';
11     std::cout << "\n\n";
12 }
13
14 int main() {
15
16     std::cout << '\n';
17
18     std::vector myVec1{1, 2, 3, 4, 5};
19     std::vector myVec2{6, 7, 8, 9};
20
21     std::span<int> dynamicSpan(myVec1);
22     std::span<int, 4> staticSpan(myVec2);
23
24     printMe(dynamicSpan);
25     printMe(staticSpan); //implicitly converted into a dynamic span
26
27     // staticSpan = dynamicSpan; ERROR
28     dynamicSpan = staticSpan;
29
30     printMe(staticSpan);
31
32     std::cout << '\n';
33
34 }
```

---

У `dynamicSpan` (строка 21) динамическая длина, в то время как у `staticSpan` (строка 22) статическая длина. Оба `std::span`'а возвращают свой размер в функции `printMe` (строка 9). Можно присвоить `std::span`-у с динамической длиной

`std::span` со статической длиной, но не наоборот. Строка 27 вызовет ошибку, но строки 7, 25 и 28 ошибок не вызывают.

```

C:\Users\seminar>staticDynamicExtentSpan.exe

container.size(): 5
1 2 3 4 5

container.size(): 4
6 7 8 9

container.size(): 4
6 7 8 9

C:\Users\seminar>

```

`std::span` со статической и динамической длиной

Одной из важных причин существования `std::span<T>` является то, что обычный массив `C` приводится к указателю при передаче в функцию, тем самым его размер теряется. Это является одной из типичных причин возникновения ошибок в `C/C++`.

## 5.2.2 Автоматический вывод размера непрерывной последовательности объектов

В отличие от массива в `C`, `std::span<T>` автоматически выводит размер непрерывной последовательности объектов.

`std::span<T>` автоматически выводит размер непрерывной последовательности объектов

---

```

1  // printSpan.cpp
2
3  #include <iostream>
4  #include <vector>
5  #include <array>
6  #include <span>
7
8  void printMe(std::span<int> container) {
9
10     std::cout << "container.size(): " << container.size() << '\n';
11     for (auto e : container) std::cout << e << ' ';
12     std::cout << "\n\n";
13 }


```

```

14
15  int main() {
16
17      std::cout << '\n';
18
19      int arr[]{1, 2, 3, 4};
20      printMe(arr);
21
22      std::vector vec{1, 2, 3, 4, 5};
23      printMe(vec);
24
25      std::array arr2{1, 2, 3, 4, 5, 6};
26      printMe(arr2);
27
28  }

```

Массив из C (строка 19), `std::vector` (строка 22) и `std::array` (строка 25) содержат значения типа `int`. Поэтому `std::span` тоже содержит значения типа `int`. Но в этом примере есть один интересный момент – для каждого контейнера `std::span` может вывести его размер (строка 10).



```

C:\Users\seminar>printSpan.exe

container.size(): 4
1 2 3 4

container.size(): 5
1 2 3 4 5

container.size(): 6
1 2 3 4 5 6

C:\Users\seminar>

```

Автоматическое определение размера `std::span`

Есть и другие пути создания `std::span`.

### 5.2.3 Создание `std::span` из указателя и размера

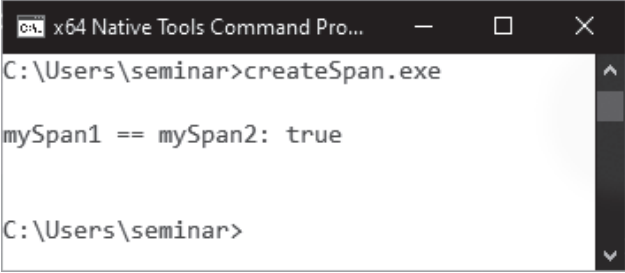
Вы можете создать `std::span` из указателя и размера последовательности объектов.

Создание `std::span`

```
1  // createSpan.cpp
2
3  #include <algorithm>
4  #include <iostream>
5  #include <span>
6  #include <vector>
7
8  int main() {
9
10     std::cout << '\n';
11     std::cout << std::boolalpha;
12
13     std::vector myVec{1, 2, 3, 4, 5};
14
15     std::span mySpan1{myVec};
16     std::span mySpan2{myVec.data(), myVec.size()};
17
18     bool spansEqual = std::equal(mySpan1.begin(), mySpan1.end(),
19                                  mySpan2.begin(), mySpan2.end());
20
21     std::cout << "mySpan1 == mySpan2: " << spansEqual << '\n';
22
23     std::cout << '\n';
24
25 }
```

---

Как вы и могли ожидать, `mySpan1`, созданный из `std::vector` (строка 15), и `mySpan2`, созданный из указателя и размера (строка 16), идентичны (строка 21).



```
C:\Users\seminar>createSpan.exe

mySpan1 == mySpan2: true

C:\Users\seminar>
```

Создание `std::span` из указателя и размера



### **std::span – это не std::string\_view и не вид**

Вы можете вспомнить, что иногда `std::span` называют видом. Но не путайте `std::span` с видом из библиотеки диапазонов и `std::string_view`<sup>1</sup>.

Вид из библиотеки диапазонов – это что-то, что вы можете применить к диапазону и выполнить некоторую операцию. Вид не владеет данными, и его время копирования, перемещения и присваивания константно.

При этом `std::span` и `std::string_view` – это не владеющие виды, и они могут работать со строками. Основная разница между ними в том, что `std::span` может изменять объекты, на которые он ссылается.

## **5.2.4 Изменение объектов, к которым происходит обращение через ссылку**

Вы можете изменить весь диапазон (`span`) или только его часть. Когда вы изменяете диапазон, то вы изменяете объекты, на которые он ссылается.

Следующая программа демонстрирует то, как можно изменить объекты из `std::vector`.

Изменение объектов, на которые указывает ссылка

---

```

1  // spanTransform.cpp
2
3  #include <algorithm>
4  #include <iostream>
5  #include <vector>
6  #include <span>
7
8  void printMe(std::span<int> container) {
9
10     std::cout << "container.size(): " << container.size() << '\n';
11     for (auto e : container) std::cout << e << ' ';
12     std::cout << "\n\n";
13 }
14
15 int main() {
16
17     std::cout << '\n';
18
19     std::vector vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
20     printMe(vec);
21

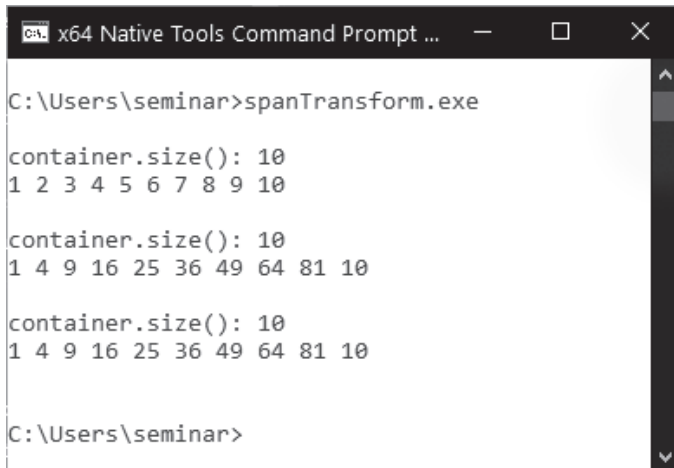
```

<sup>1</sup> <https://www.modernescpp.com/index.php/c-17-what-s-new-in-the-library>.

```
22     std::span span1(vec);
23     std::span span2{span1.subspan(1, span1.size() - 2)};
24
25
26     std::transform(span2.begin(), span2.end(),
27                   span2.begin(),
28                   [](int i){ return i * i; });
29
30
31     printMe(vec);
32     printMe(span1);
33
34 }
```

---

Здесь `span1` ссылается на `std::vector vec` (строка 22). В отличие от этого, `span2` ссылается на элементы `vec`, за исключением первого и последнего (строка 23). Соответственно, замена каждого элемента на его возведение в квадрат (строка 26) касается только этих элементов.



```
C:\Users\seminar>spanTransform.exe

container.size(): 10
1 2 3 4 5 6 7 8 9 10

container.size(): 10
1 4 9 16 25 36 49 64 81 10

container.size(): 10
1 4 9 16 25 36 49 64 81 10

C:\Users\seminar>
```

Изменение объектов, на которые указывает ссылка

Есть целый ряд удобных функций, с помощью которых можно обращаться к элементам `std::span`.

### 5.2.5 Обращение к элементам `std::span`

Следующая таблица приводит список функций для доступа к элементам `std::span`.



| Функция                                                                                | Описание                                                                    |
|----------------------------------------------------------------------------------------|-----------------------------------------------------------------------------|
| <code>sp.front()</code>                                                                | Обращается к первому элементу                                               |
| <code>sp.back()</code>                                                                 | Обращается к последнему элементу                                            |
| <code>sp[i]</code>                                                                     | Обращается к элементу <i>i</i>                                              |
| <code>sp.data()</code>                                                                 | Возвращает указатель на первый элемент                                      |
| <code>sp.size()</code>                                                                 | Возвращает количество элементов в последовательности                        |
| <code>sp.size_bytes()</code>                                                           | Возвращает размер последовательности в байтах                               |
| <code>sp.empty()</code>                                                                | Возвращает true, если последовательность пуста                              |
| <code>sp.first&lt;count&gt;()</code><br><code>sp.first(count)</code>                   | Возвращает подпоследовательность из первых count элементов                  |
| <code>sp.last&lt;count&gt;()</code><br><code>sp.last(count)</code>                     | Возвращает подпоследовательность из последних count элементов               |
| <code>sp.subspan&lt;first, count&gt;()</code><br><code>sp.subspan(first, count)</code> | Возвращает заданную подпоследовательность (от first) длиной count элементов |

Программа `subspan.cpp` демонстрирует использование функции `subspan`.

Использование функции `subspan`

---

```

1  // subspan.cpp
2
3  #include <iostream>
4  #include <numeric>
5  #include <span>
6  #include <vector>
7
8  int main() {
9
10     std::cout << '\n';
11
12     std::vector<int> myVec(20);
13     std::iota(myVec.begin(), myVec.end(), 0);
14     for (auto v: myVec) std::cout << v << " ";
15
16     std::cout << "\n\n";
17
18     std::span<int> mySpan(myVec);
19     auto length = mySpan.size();
20
21     std::size_t count = 5;
```

```

22  for (std::size_t first = 0; first <= (length - count); first += count ) {
23      for (auto ele: mySpan.subspan(first, count)) std::cout << ele << " ";
24      std::cout << '\n';
25  }
26
27  }

```

Строка 13 заполняет вектор числами от 0 до 19 (строка 13) при помощи алгоритма `std::iota`<sup>1</sup>. Этот вектор далее применяется для инициализации `std::span` (строка 18). Наконец, цикл `for` (строка 22) использует функцию `subspan` для создания всех подпоследовательностей, начинающихся с элемента `first` и содержащих `count` элементов, пока не будет перебран весь `mySpan`.

Использование метода `subspan`

Кириан Хеленбергер напомнил мне о специальном случае использования `std::span` – о постоянном диапазоне изменяемых элементов.

## 5.2.6 Постоянный диапазон изменяемых элементов

Обычно `std::vector` и `std::string` моделируют изменяемый диапазон изменяемых элементов. Когда вы их объявляете как `const`, то получаете неизменяемый диапазон неизменяемых элементов. Таким образом, вы не можете создать неизменяемый диапазон изменяемых элементов. И здесь нам на помощь приходит `std::span`. `std::span` моделирует неизменяемый диапазон изменяемых элементов: `std::span<T>`. Следующая таблица сводит вместе все возможные вариации использования `std::span`.

|                              | Изменяемые элементы               | Неизменяемые элементы                                                            |
|------------------------------|-----------------------------------|----------------------------------------------------------------------------------|
| <b>Изменяемый диапазон</b>   | <code>std::vector&lt;T&gt;</code> |                                                                                  |
| <b>Неизменяемый диапазон</b> | <code>std::span&lt;T&gt;</code>   | <code>const std::vector&lt;T&gt;</code><br><code>std::span&lt;const T&gt;</code> |

Программа `constRangeModifiableElements.cpp` показывает каждую комбинацию.

<sup>1</sup> <https://en.cppreference.com/w/cpp/algorithm/iota>.

Неизменяемый/изменяемый диапазон неизменяемых/изменяемых элементов

---

```

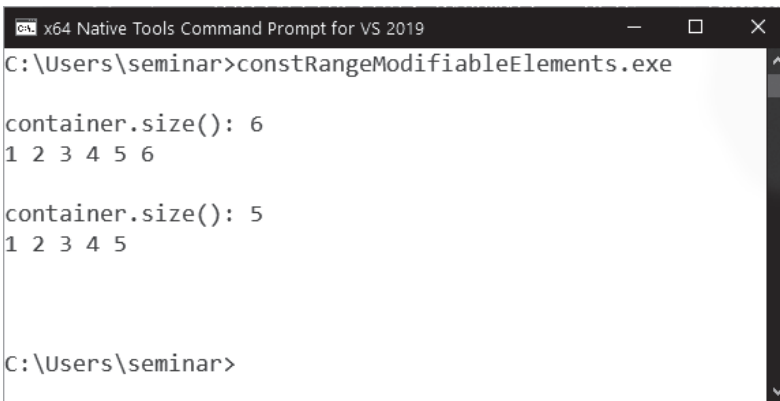
1  // constRangeModifiableElements.cpp
2
3  #include <iostream>
4  #include <span>
5  #include <vector>
6
7  void printMe(std::span<int> container) {
8
9      std::cout << "container.size(): " << container.size() << '\n';
10     for (auto e : container) std::cout << e << ' ';
11     std::cout << "\n\n";
12 }
13
14 int main() {
15
16     std::cout << '\n';
17
18     std::vector<int> origVec{1, 2, 2, 4, 5};
19
20     // Modifiable range of modifiable elements
21     std::vector<int> dynamVec = origVec;
22     dynamVec[2] = 3;
23     dynamVec.push_back(6);
24     printMe(dynamVec);
25
26     // Constant range of constant elements
27     const std::vector<int> constVec = origVec;
28     // constVec[2] = 3;          ERROR
29     // constVec.push_back(6);    ERROR
30     std::span<const int> constSpan(origVec);
31     // constSpan[2] = 3;        ERROR
32
33     // Constant range of modifiable elements
34     std::span<int> dynamSpan{origVec};
35     dynamSpan[2] = 3;
36     printMe(dynamSpan);
37

```

```
38     std::cout << '\n';  
39  
40 }
```

---

Вектор `dynamicVec` (строка 21) – это модифицируемый диапазон модифицируемых элементов. Но это не относится к `constVec` (строка 27). В случае `constVec` нельзя изменить ни элементы, ни размер. `constSpan` (строка 30) ведет себя аналогично. `dynamicSpan` соответствует неизменяемому диапазону с изменяемыми элементами.



```
C:\Users\seminar>constRangeModifiableElements.exe  
  
container.size(): 6  
1 2 3 4 5 6  
  
container.size(): 5  
1 2 3 4 5  
  
C:\Users\seminar>
```

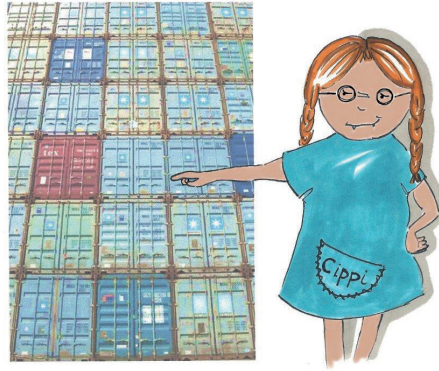
Неизменяемый/изменяемый диапазон неизменяемых/изменяемых элементов



### Краткая информация

- ♦ `std::span` – это объект, который ссылается на непрерывную последовательность объектов. `std::span`, иногда называемый видом, никогда не является владельцем и поэтому не выделяет память. Непрерывная последовательность объектов может быть массивом C, указателем вместе с размером, `std::array`, `std::vector` или `std::string`.
- ♦ В отличие от массива в C, `std::span` автоматически выводит размер последовательности объектов, на которую он ссылается.
- ♦ Когда `std::span` изменяет свои элементы, то объекты, на которые он ссылается, также изменяются.

## 5.3 Улучшения контейнеров



Сиппи изучает контейнеры

В стандарте C++20 появилось много улучшений, относящихся к контейнерам из стандартной библиотеки шаблонов (STL). Прежде всего у `std::vector` и `std::string` теперь есть `constexpr`-конструкторы, поэтому они могут использоваться во время компиляции. Все контейнеры поддерживают последовательное удаление из контейнера, а ассоциативные контейнеры поддерживают метод `contains`. Кроме того, `std::string` позволяет проверить строку на наличие префикса или суффикса.

### 5.3.1 Контейнеры и алгоритмы со спецификатором `constexpr`

В стандарте C++20 поддерживаются `constexpr`-контейнеры `std::vector` и `std::string`, где `constexpr` означает, что методы обоих контейнеров могут быть вызваны во время компиляции. Кроме того, более 100 алгоритмов<sup>1</sup> из стандартной библиотеки шаблонов теперь стали `constexpr`.

Соответственно, теперь вы можете сортировать `std::vector` из переменных типа `int` во время компиляции.

Сортировка `std::vector` во время компиляции

---

```
1 // constexprVector.cpp
2
3 #include <algorithm>
4 #include <iostream>
5 #include <vector>
6
```

<sup>1</sup> <https://en.cppreference.com/w/cpp/algorithm>.

```
7  constexpr int maxElement() {
8      std::vector myVec = {1, 2, 4, 3};
9      std::sort(myVec.begin(), myVec.end());
10     return myVec.back();
11 }
12 int main() {
13
14     std::cout << '\n';
15
16     constexpr int maxValue = maxElement();
17     std::cout << "maxValue: " << maxValue << '\n';
18
19     constexpr int maxValue2 = [] {
20         std::vector myVec = {1, 2, 4, 3};
21         std::sort(myVec.begin(), myVec.end()) ;
22         return myVec.back();
23     }();
24
25     std::cout << "maxValue2: " << maxValue2 << '\n';
26
27     std::cout << '\n';
28
29 }
```

---

Два контейнера `std::vector` (строки 8 и 20) сортируются во время компиляции при помощи функций, объявленных как `constexpr`. В первом случае функция `maxElement` возвращает последний элемент вектора `myVec`, являющийся его максимальным элементом. Во втором случае я использую сразу же вызываемую лямбду, которая объявлена как `constexpr`.

```
> include <algorithm>
> include <cstdint>
> include <vector>
>
> constexpr int maxElement() {
>     std::vector myVec = {1, 2, 4, 3};
>     std::sort(myVec.begin(), myVec.end());
>     return myVec.back();
> }
> int main() {
>     std::cout << '\n';
> }
```

Сортировка `std::vector` во время компиляции

### 5.3.2 `std::array`

C++20 предоставляет два способа создания массивов: `std::to_array` создает `std::array`, а `std::make_shared` позволяет создать `std::shared_ptr` массив.

### 5.3.2.1 std::to\_array

std::to\_array позволяет создать std::array из существующего одномерного массива. Элементы созданного std::array инициализируются путем копирования элементов существующего одномерного массива.

Одномерный массив может быть строкой C, std::initialize\_list или же одномерным массивом std::pair. Следующий пример взят с ресурса [cppreference.com/to\\_array](https://en.cppreference.com/to_array)<sup>1</sup>.

Создание std::array из различных одномерных массивов

---

```

1 // toArray.cpp
2
3 #include <iostream>
4 #include <utility>
5 #include <array>
6 #include <memory>
7
8 int main() {
9
10     std::cout << '\n';
11
12     auto arr1 = std::to_array("A simple test");
13     for (auto a: arr1) std::cout << a;
14     std::cout << "\n\n";
15
16     auto arr2 = std::to_array({1, 2, 3, 4, 5});
17     for (auto a: arr2) std::cout << a;
18     std::cout << "\n\n";
19
20     auto arr3 = std::to_array<double>({0, 1, 3});
21     for (auto a: arr3) std::cout << a;
22     std::cout << '\n';
23     std::cout << "typeid(arr3[0]).name(): " << typeid(arr3[0]).name() << '\n';
24     std::cout << '\n';
25
26     auto arr4 = std::to_array<std::pair<int, double>>({ {1, 0.0}, {2, 5.1},
27                                                         {3, 5.1} });
28     for (auto p: arr4) {
29         std::cout << "(" << p.first << ", " << p.second << ")" << '\n';
30     }
31

```

---

<sup>1</sup> [https://en.cppreference.com/w/cpp/container/array/to\\_array](https://en.cppreference.com/w/cpp/container/array/to_array).

```
32     std::cout << "\n\n";
33
34 }
```

---

Я создал `std::array` из C-строки (строка 12), из `std::initializer_list` (строки 16 и 20), а `std::initialize_list` – из `std::pair` (строка 26). В общем случае компилятор может вывести тип `std::array`. Можно и явно указать тип (строки 20 и 26).

```
A simple test

12345

013
typeid(arr3[0]).name(): d

(1, 0)
(2, 5.1)
(3, 5.1)
```

Создание `std::array` из различных одномерных массивов

### 5.3.2.2 `std::make_shared`

Начиная со стандарта C++11 в C++ появилась поддержка создания `std::shared_ptr` при помощи фабричной функции `std::make_shared`<sup>1</sup>. Начиная с C++20 эта фабричная функция поддерживает создание массивов из `std::shared_ptr`.

- `std::shared_ptr<double[]> shar = std::make_shared<double[]>(1024);` создает `std::shared_ptr` с 1024 значениями типа `double`, проинициализированными по умолчанию;
- `std::shared_ptr<double[]> shar = std::make_shared<double[]>(1024, 1.0);` создает `std::shared_ptr` с 1024 значениями типа `double`, проинициализированными значениями, равными 1,0.

## 5.3.3 Последовательное удаление из контейнеров

До выхода стандарта C++20 удаление элементов из контейнера было слишком сложным, и сейчас я покажу почему.

### 5.3.3.1 Идиома `erase-remove`

Удаление элемента из контейнера кажется очень простым. В случае `std::vector` вы можете использовать `std::remove_if`.

---

<sup>1</sup> [https://en.cppreference.com/w/cpp/memory/shared\\_ptr/make\\_shared](https://en.cppreference.com/w/cpp/memory/shared_ptr/make_shared).



Использование `std::remove_if` для удаления элемента из контейнера

---

```

1 // removeElements.cpp
2
3 #include <algorithm>
4 #include <iostream>
5 #include <vector>
6
7 int main() {
8
9     std::cout << '\n';
10
11     std::vector myVec{-2, 3, -5, 10, 3, 0, -5 };
12
13     for (auto ele: myVec) std::cout << ele << " ";
14     std::cout << "\n\n";
15
16     std::remove_if(myVec.begin(), myVec.end(), [](int ele){ return ele < 0; });
17     for (auto ele: myVec) std::cout << ele << " ";
18
19     std::cout << "\n\n";
20
21 }
```

---

Программа `removeElements.cpp` удаляет из `std::vector` все элементы, которые меньше нуля. Вроде бы ничего сложного. Возможно, и нет; теперь вы попадаете в ловушку, хорошо известную многим опытным C++-программистам.

```

2
3 #include <algorithm>
4 #include <iostream>
5 #include <vector>
6
7 int main() {
8
9     std::cout << '\n';
10
11     std::vector myVec{-2, 3, -5, 10, 3, 0, -5 };
12
13     for (auto ele: myVec) std::cout << ele << " ";
14     std::cout << "\n\n";
15
16     std::remove_if(myVec.begin(), myVec.end(), [](int ele){ return ele < 0; });
17     for (auto ele: myVec) std::cout << ele << " ";
18
19     std::cout << "\n\n";
20 }
```

Использование `std::remove_if` для удаления элемента из контейнера

На самом деле `std::remove_if` (строка 16) не удаляет ничего. После нее `std::vector` содержит столько же элементов. Но алгоритм возвращает логический конец измененного контейнера.

Для изменения контейнера вы должны применить этот логический конец к контейнеру.

Применение идиомы `erase-remove` к контейнеру

---

```
1  // eraseRemoveElements.cpp
2
3  #include <algorithm>
4  #include <iostream>
5  #include <vector>
6
7  int main() {
8
9      std::cout << '\n';
10
11     std::vector myVec{-2, 3, -5, 10, 3, 0, -5 };
12
13     for (auto ele: myVec) std::cout << ele << " ";
14     std::cout << "\n\n";
15
16     auto newEnd = std::remove_if(myVec.begin(), myVec.end(),
17                                 [](int ele){ return ele < 0; });
18     myVec.erase(newEnd, myVec.end());
19     // myVec.erase(std::remove_if(myVec.begin(), myVec.end(),
20     //                             [](int ele){ return ele < 0; }), myVec.end());
21     for (auto ele: myVec) std::cout << ele << " ";
22
23     std::cout << "\n\n";
24
25 }
```

---

Строка 16 возвращает новый логический конец `newEnd` контейнера `myVec`. Этот новый логический конец применяется в строке 18 для удаления всех элементов из `myVec` начиная с `newEnd`. Когда вы применяете функции `remove` и `erase` в одном выражении, таком как в строке 19, вы точно понимаете, почему это называется идиомой `erase-remove`.

```

rainer@seminar:~> eraseRemoveElements

-2 3 -5 10 3 0 -5

3 10 3 0

rainer@seminar:~>

```

Применение идиомы erase-remove к контейнеру

Благодаря новым функциям `erase` и `erase_if` в C++20 удаление элементов становится гораздо более удобным.

### 5.3.3.2 `erase` и `erase_if` в C++20

При помощи `erase` и `erase_if` вы можете непосредственно работать с контейнерами. Для сравнения, ранее показанная идиома `erase-remove` сложна: она требует двух итераций.

Давайте посмотрим, что новые функции `erase` и `erase_if` значат на практике. Следующая программа удаляет элементы из двух контейнеров.

Удаление элементов из контейнера

---

```

1  // eraseCpp20.cpp
2
3  #include <iostream>
4  #include <numeric>
5  #include <deque>
6  #include <list>
7  #include <string>
8  #include <vector>
9
10 template <typename Cont>
11 void eraseVal(Cont& cont, int val) {
12     std::erase(cont, val);
13 }
14
15 template <typename Cont, typename Pred>
16 void erasePredicate(Cont& cont, Pred pred) {
17     std::erase_if(cont, pred);
18 }
19

```

```
20 template <typename Cont>
21 void printContainer(Cont& cont) {
22     for (auto c: cont) std::cout << c << " ";
23     std::cout << '\n';
24 }
25
26 template <typename Cont>
27 void doAll(Cont& cont) {
28     printContainer(cont);
29     eraseVal(cont, 5);
30     printContainer(cont);
31     erasePredicate(cont, [](auto i) { return i >= 3; });
32     printContainer(cont);
33 }
34
35 int main() {
36
37     std::cout << '\n';
38
39     std::string str{"A Sentence with an E."};
40     std::cout << "str: " << str << '\n';
41     std::erase(str, 'e');
42     std::cout << "str: " << str << '\n';
43     std::erase_if( str, [](char c){ return std::isupper(c); });
44     std::cout << "str: " << str << '\n';
45
46     std::cout << "\nstd::vector " << '\n';
47     std::vector vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
48     doAll(vec);
49
50     std::cout << "\nstd::deque " << '\n';
51     std::deque deq{1, 2, 3, 4, 5, 6, 7, 8, 9};
52     doAll(deq);
53
54     std::cout << "\nstd::list" << '\n';
55     std::list lst{1, 2, 3, 4, 5, 6, 7, 8, 9};
56     doAll(lst);
57
58 }
```

---

Строка 41 удаляет все символы 'e' из заданной строки `str`. Строка 43 применяет лямбду к этой же строке и удаляет все заглавные символы.

В оставшейся части программы удаляются элементы из следующих контейнеров: `std::vector` (строка 47), `std::deque` (строка 51) и `std::list` (строка 55). Для каждого контейнера применяется шаблонная функция `doAll` (строка 26). Функция `doAll` удаляет элемент 5 и все элементы, равные или большие 3. Шаблонная функция `eraseVal` (строка 10) использует новую функцию `erase`. Шаблонная функция `erasePredicat` (строка 15) использует новую функцию `erase_if`.

```

C:\Users\seminar>eraseCpp20.exe

str: A Sentence with an E.
str: A Sntnc with an E.
str: ntnc with an .

std::vector
1 2 3 4 5 6 7 8 9
1 2 3 4 6 7 8 9
1 2

std::deque
1 2 3 4 5 6 7 8 9
1 2 3 4 6 7 8 9
1 2

std::list
1 2 3 4 5 6 7 8 9
1 2 3 4 6 7 8 9
1 2

C:\Users\seminar>

```

Применение новых функций `erase` и `erase_if`

Новые функции `erase` и `erase_if` могут применяться ко всем контейнерам из стандартной библиотеки шаблонов. Это неверно для функции `contains`, которая может применяться только к ассоциативным контейнерам.

### 5.3.4 `contains` для ассоциативных контейнеров

Благодаря функции `contains` вы можете легко проверить, содержится ли элемент в ассоциативном контейнере. Вы можете сказать, что вы могли это делать и ранее при помощи `find` или `count`.

На самом деле эти функции не очень дружелюбны для новичков и обладают своими недостатками.

Проверка на наличие элемента в контейнере

---

```
1 // checkExistence.cpp
2
3 #include <set>
4 #include <iostream>
5
6 int main() {
7
8     std::cout << '\n';
9
10    std::set mySet{3, 2, 1};
11    if (mySet.find(2) != mySet.end()) {
12        std::cout << "2 inside" << '\n';
13    }
14
15    std::multiset myMultiSet{3, 2, 1, 2};
16    if (myMultiSet.count(2)) {
17        std::cout << "2 inside" << '\n';
18    }
19
20    std::cout << '\n';
21
22 }
```

---

Эта программа дает вполне ожидаемый результат.

```
1 // checkExistence.cpp
2 // checkExistence.cpp
3
4 int main() {
5
6     std::cout << '\n';
7
8     std::set mySet{3, 2, 1};
9     if (mySet.find(2) != mySet.end()) {
10         std::cout << "2 inside" << '\n';
11     }
12
13     std::multiset myMultiSet{3, 2, 1, 2};
14     if (myMultiSet.count(2)) {
15         std::cout << "2 inside" << '\n';
16     }
17
18     std::cout << '\n';
19 }
```

Использование `find` и `count` для проверки, содержит ли контейнер заданный элемент

Существуют определенные проблемы с обоими этими вызовами. Вызов `find` (строка 11) слишком многословен. То же самое относится и к вызову `count` (строка 16). Кроме того, у `count` есть проблемы с быстродействием. Когда вы хотите узнать, содержится ли элемент в контейнере, то вам следует остановиться, как только вы найдете первый элемент, а не считать до самого конца. В этом конкретном примере `myMultiSet.count(2)` возвращает 2.

В отличие от `find` и `count`, метод `contains` из C++20 удобен для использования.

## Функция contains в C++20

---

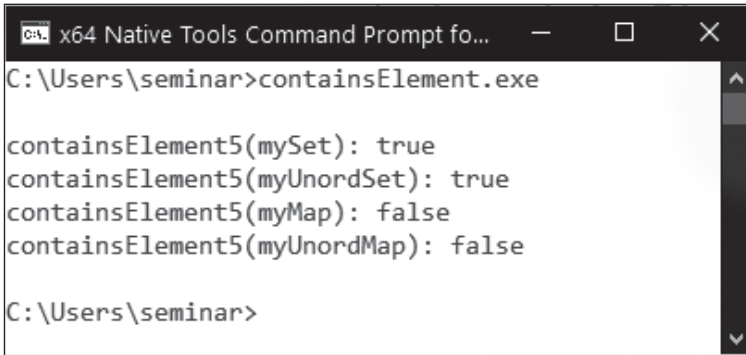
```

1  // containsElement.cpp
2
3  #include <iostream>
4  #include <set>
5  #include <map>
6  #include <unordered_set>
7  #include <unordered_map>
8
9  template <typename AssocCont>
10 bool containsElement5(const AssocCont& assocCont) {
11     return assocCont.contains(5);
12 }
13
14 int main() {
15
16     std::cout << std::boolalpha;
17
18     std::cout << '\n';
19
20     std::set<int> mySet{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
21     std::cout << "containsElement5(mySet): " << containsElement5(mySet);
22
23     std::cout << '\n';
24
25     std::unordered_set<int> myUnordSet{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
26     std::cout << "containsElement5(myUnordSet): "
27               << containsElement5(myUnordSet);
28     std::cout << '\n';
29
30     std::map<int, std::string> myMap{ {1, "red"}, {2, "blue"}, {3, "green"} };
31     std::cout << "containsElement5(myMap): " << containsElement5(myMap);
32
33     std::cout << '\n';
34
35     std::unordered_map<int, std::string> myUnordMap{ {1, "red"},
36                                                     {2, "blue"}, {3, "green"} };
37     std::cout << "containsElement5(myUnordMap): "
38               << containsElement5(myUnordMap);
39     std::cout << '\n';
40
41 }

```

---

К этому примеру даже нечего больше добавить. Шаблонная функция возвращает true, если переданный ассоциативный контейнер содержит значение 5. В моем примере я использовал только следующие ассоциативные контейнеры: `std::set`, `std::unordered_set`, `std::map` и `std::unordered_map`. Ни один из них не может использовать заданный ключ более чем один раз.



```

C:\Users\seminar>containsElement.exe

containsElement5(mySet): true
containsElement5(myUnordSet): true
containsElement5(myMap): false
containsElement5(myUnordMap): false

C:\Users\seminar>

```

Использование нового метода `contains`

### 5.3.5 Проверка строки на наличие префикса и суффикса

У `std::string` появились два новых метода `starts_with` и `ends_with`. Они позволяют проверить, начинается ли `std::string` с заданной подстроки и заканчивается ли она на заданную подстроку.

Проверка строки на наличие префикса и суффикса

```

1  // stringStartsWithEndsWith.cpp
2
3  #include <iostream>
4  #include <string_view>
5  #include <string>
6
7  template <typename PrefixType>
8  void startsWith(const std::string& str, PrefixType prefix) {
9      std::cout << "                starts with " << prefix << ": "
10         << str.starts_with(prefix) << '\n';
11  }
12
13  template <typename SuffixType>
14  void endsWith(const std::string& str, SuffixType suffix) {
15      std::cout << "                ends with " << suffix << ": "
16         << str.ends_with(suffix) << '\n';
17  }
18

```



---

```
19  int main() {
20
21      std::cout << '\n';
22
23      std::cout << std::boolalpha;
24
25      std::string helloWorld("Hello World");
26
27      std::cout << helloWorld << '\n';
28
29      startsWith(helloWorld, helloWorld);
30
31      startsWith(helloWorld, std::string_view("Hello"));
32
33      startsWith(helloWorld, 'H');
34
35      std::cout << "\n\n";
36
37      std::cout << helloWorld << '\n';
38
39      endsWith(helloWorld, helloWorld);
40
41      endsWith(helloWorld, std::string_view("World"));
42
43      endsWith(helloWorld, 'd');
44
45  }
```

---

Оба метода `starts_with` и `ends_with` являются предикатами, т. е. возвращают логическое значение. Их можно вызывать для `std::string` (строки 29 и 39), `std::string_view` (строки 31 и 41) и `char` (строки 33 и 43).

```
Hello World
    starts with Hello World: true
    starts with Hello: true
    starts with H: true

Hello World
    ends with Hello World: true
    ends with World: true
    ends with d: true
```

Проверка строки на наличие префикса и суффикса

### Краткая информация

- ♦ У `std::vector` и `std::string` теперь есть `constexpr`-конструкторы, и поэтому они могут быть созданы во время компиляции. Благодаря `constexpr`-алгоритмам из стандартной библиотеки шаблонов (STL) вы можете манипулировать ими во время компиляции.
- ♦ C++20 предоставляет два новых способа создания массивов. `std::to_array` создает `std::array`, и `std::make_shared` позволяет создавать `std::shared_ptr` для обычного массива C.
- ♦ Новые алгоритмы `std::erase` и `std::erase_if` могут быть использованы для удаления заданных элементов (`erase`) или элементов, удовлетворяющих заданному предикату (`erase_if`) из произвольного контейнера в STL.
- ♦ Благодаря методу `contains` вы можете проверить, содержит ли произвольный ассоциативный контейнер заданный элемент (ключ).
- ♦ `std::string` поддерживает два новых метода `starts_with` и `ends_with` для проверки, начинается или заканчивается строка с заданным префиксом или суффиксом.

## 5.4 Арифметические функции



Сиппи изучает арифметику

Сравнение знаковых и беззнаковых целых чисел – это частая причина для непредвиденного поведения программы и, конечно же, возникновения ошибок. Благодаря новым безопасным функциям сравнения для целых чисел `std::cmp_*` этот источник ошибок исчез. Также стандарт C++20 вводит ряд математических констант, таких как  $e$ ,  $\pi$  или  $\phi$ , и при помощи функций `std::midpoint` и `std::lerp` вы можете вычислить среднее между двумя числами или же произвести линейную интерполяцию. Новые функции для манипулирования битами позволяют обращаться и изменять отдельные биты или последовательности битов.

### 5.4.1 Безопасное сравнение целых чисел

Когда вы сравниваете целые числа со знаком и без знака, то вы можете получить совсем не тот результат, который ожидаете. Благодаря шести функциям `std::cmp_*` теперь для этой проблемы в C++20 есть решение. Для объяснения, зачем это нужно, я сначала покажу небезопасный вариант.



#### Целое значение против целочисленного

Термины целочисленный (*integral*) и целый (*integer*) в C++ являются синонимами. Вот формулировка из стандарта по типам: «Типы `bool`, `char`, `char8_t`, `char16_t`, `char32_t`, `wchar_t` и `signed` и `unsigned` вместе называются целочисленными типами. Синонимом целочисленного типа является целый тип».

#### 5.4.1.1 Небезопасное сравнение

Для названия `unsafeComparison.cpp` есть серьезные основания.

## Небезопасное сравнение целых чисел

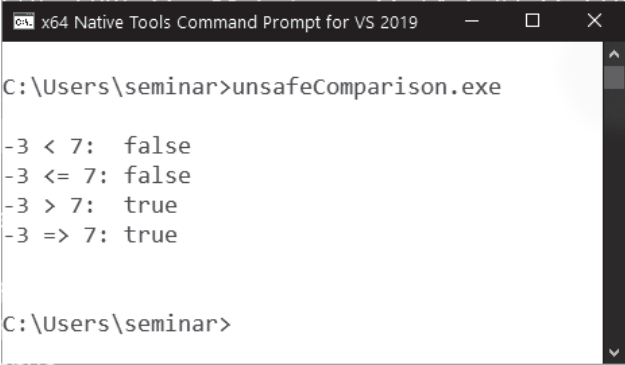
---

```

1  // unsafeComparison.cpp
2
3  #include <iostream>
4
5  int main() {
6
7      std::cout << '\n';
8
9      std::cout << std::boolalpha;
10
11     int x = -3;
12     unsigned int y = 7;
13
14     std::cout << "-3 < 7: " << (x < y) << '\n';
15     std::cout << "-3 <= 7: " << (x <= y) << '\n';
16     std::cout << "-3 > 7: " << (x > y) << '\n';
17     std::cout << "-3 == 7: " << (x == y) << '\n';
18
19     std::cout << '\n';
20
21 }
```

---

Если выполнить эту программу, то результат может не соответствовать ожиданиям.



```

C:\Users\seminar>unsafeComparison.exe

-3 < 7: false
-3 <= 7: false
-3 > 7: true
-3 == 7: true

C:\Users\seminar>
```

Сюрпризы с небезопасным сравнением целых чисел

Если проанализировать вывод этой программы, то можно заметить, что -3 больше 7. Вы, наверное, знаете причину. Я сравнил знаковое x (строка 11) с без-

знаковым `y` (строка 12). Что при этом на самом деле происходит? Следующая программа дает ответ на этот вопрос.

Объяснение небезопасности сравнения целых чисел

---

```

1 // unsafeComparison2.cpp
2
3 int main() {
4     int x = -3;
5     unsigned int y = 7;
6
7     bool val = x < y;
8     static_assert(static_cast<unsigned int>(-3) == 4'294'967'293);
9 }
```

---

В этом примере я хочу обратить ваше внимание на оператор «меньше чем». C++ Insights<sup>1</sup> дает мне следующий вывод:

---

```

1 // unsafeComparison2.cpp
2
3 int main() {
4     int x = -3;
5     unsigned int y = 7;
6
7     bool val = x < y;
```

---

Анализ небезопасного сравнения

Вот что происходит:

- компилятор переводит выражение `x < y` (строка 7) в `static_cast<unsigned int>(x) < y`. В частности, `signed x` переводится в `unsigned int`;
- из-за этого преобразования `-3` заменяется значением `4'294'967'293`;
- `4'294'967'293` равно `-3` по модулю  $2^{32}$ ;
- `32` – это количество битов в `unsigned int` на C++ Insights.

Благодаря вводу стандарта C++20 у нас теперь есть безопасное сравнение целых чисел.

#### 5.4.1.2 Безопасное сравнение целых чисел

C++20 поддерживает шесть функций сравнения для целых чисел:

| Функция сравнения               | Значение          |
|---------------------------------|-------------------|
| <code>std::cmp_equal</code>     | <code>==</code>   |
| <code>std::cmp_not_equal</code> | <code>!=</code>   |
| <code>std::cmp_less</code>      | <code>&lt;</code> |

<sup>1</sup> <https://cppinsights.io/s/62732a01>.

| Функция сравнения                   | Значение           |
|-------------------------------------|--------------------|
| <code>std::cmp_less_equal</code>    | <code>&lt;=</code> |
| <code>std::cmp_greater</code>       | <code>&gt;</code>  |
| <code>std::cmp_greater_equal</code> | <code>&gt;=</code> |

Благодаря этим шести функциям сравнения я могу легко переписать пример `unsafeComparison.cpp` в программу `safeComparison.cpp`. Новые функции для сравнения описаны в заголовочном файле `<utility>`.

Безопасное сравнение целых чисел

---

```
// safeComparison.cpp
```

```
#include <iostream>
```

```
#include <utility>
```

```
int main() {
```

```
    std::cout << '\n';
```

```
    std::cout << std::boolalpha;
```

```
    int x = -3;
```

```
    unsigned int y = 7;
```

```
    std::cout << "-3 == 7: " << std::cmp_equal(x, y) << '\n';
```

```
    std::cout << "-3 != 7: " << std::cmp_not_equal(x, y) << '\n';
```

```
    std::cout << "-3 < 7: " << std::cmp_less(x, y) << '\n';
```

```
    std::cout << "-3 <= 7: " << std::cmp_less_equal(x, y) << '\n';
```

```
    std::cout << "-3 > 7: " << std::cmp_greater(x, y) << '\n';
```

```
    std::cout << "-3 >= 7: " << std::cmp_greater_equal(x, y) << '\n';
```

```
    std::cout << '\n';
```

```
}
```

---

Кроме того, я использовал операторы «равно» и «не равно».

```
-3 == 7: false
-3 != 7: true
-3 < 7: true
-3 <= 7: true
-3 > 7: false
-3 => 7: false
```

Безопасное сравнение

Вызов безопасной функции с нецелым аргументом, например `double`, приводит к ошибке компиляции.

Безопасное сравнение `unsigned int` и `double`

```
-3 == 7: false
-3 != 7: true
-3 < 7: true
```

С другой стороны, вы можете сравнить `double` и `unsigned int` классическим способом. Программа `classicalComparison.cpp` использует классическое сравнение `double` и `unsigned int`.

Классическое сравнение `double` и `unsigned int`

---

```
// classicalComparison.cpp
```

```
int main() {

    double x = -3.5;
    unsigned int y = 7;

    auto res = x < y;    // true

}
```

---

Это работает: `unsigned int` переводится<sup>1</sup> в `double`. C++ Insights<sup>2</sup> показывает правду.

```
int main()
{
    double x = -3.5;
    unsigned int y = 7;
    bool res = x < static_cast<double>(y);
}
```

Перевод числа в `double`

## 5.4.2 Математические константы

Прежде всего, чтобы использовать константы, требуется подключить заголовочный файл `<numbers>`. Все константы расположены в пространстве имен `std::numbers`. Следующая таблица дает вам обзорную информацию.

Математические константы

| Математическая константа              | Описание                                |
|---------------------------------------|-----------------------------------------|
| <code>std::numbers::e</code>          | $e$                                     |
| <code>std::numbers::log2e</code>      | $\log_2 e$                              |
| <code>std::numbers::log10e</code>     | $\log_{10} e$                           |
| <code>std::numbers::pi</code>         | $\pi$                                   |
| <code>std::numbers::inv_pi</code>     | $\frac{1}{\pi}$                         |
| <code>std::numbers::inv_sqrtpi</code> | $\frac{1}{\sqrt{\pi}}$                  |
| <code>std::numbers::ln2</code>        | $\ln 2$                                 |
| <code>std::numbers::ln10</code>       | $\ln 10$                                |
| <code>std::numbers::sqrt2</code>      | $\sqrt{2}$                              |
| <code>std::numbers::sqrt3</code>      | $\sqrt{3}$                              |
| <code>std::numbers::inv_sqrt3</code>  | $\frac{1}{\sqrt{3}}$                    |
| <code>std::numbers::egamma</code>     | Константа Эйлера–Маскерони <sup>3</sup> |
| <code>std::numbers::phi</code>        | $\phi$                                  |

<sup>1</sup> [https://en.cppreference.com/w/cpp/language/implicit\\_conversion](https://en.cppreference.com/w/cpp/language/implicit_conversion).

<sup>2</sup> <https://cppinsights.io/s/44216566>.

<sup>3</sup> [https://en.wikipedia.org/wiki/Euler%E2%80%93Mascheroni\\_constant](https://en.wikipedia.org/wiki/Euler%E2%80%93Mascheroni_constant).



Программа `mathematicConstants.cpp` выводит эти константы.

Математические константы

---

```
// mathematicConstants.cpp
```

```
#include <iomanip>
#include <iostream>
#include <numbers>

int main() {

    std::cout << '\n';

    std::cout<< std::setprecision(10);

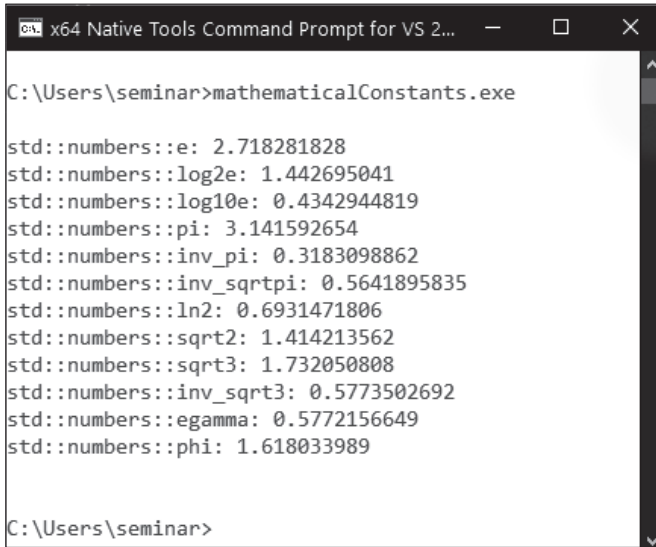
    std::cout << "std::numbers::e: " << std::numbers::e << '\n';
    std::cout << "std::numbers::log2e: " << std::numbers::log2e << '\n';
    std::cout << "std::numbers::log10e: " << std::numbers::log10e << '\n';
    std::cout << "std::numbers::pi: " << std::numbers::pi << '\n';
    std::cout << "std::numbers::inv_pi: " << std::numbers::inv_pi << '\n';
    std::cout << "std::numbers::inv_sqrtpi: " << std::numbers::inv_sqrtpi << '\n';
    std::cout << "std::numbers::ln2: " << std::numbers::ln2 << '\n';
    std::cout << "std::numbers::sqrt2: " << std::numbers::sqrt2 << '\n';
    std::cout << "std::numbers::sqrt3: " << std::numbers::sqrt3 << '\n';
    std::cout << "std::numbers::inv_sqrt3: " << std::numbers::inv_sqrt3 << '\n';
    std::cout << "std::numbers::egamma: " << std::numbers::egamma << '\n';
    std::cout << "std::numbers::phi: " << std::numbers::phi << '\n';

    std::cout << '\n';

}
```

---

Вывод этой программы при использовании компилятора MSVC.



```

C:\Users\seminar>mathematicalConstants.exe

std::numbers::e: 2.718281828
std::numbers::log2e: 1.442695041
std::numbers::log10e: 0.4342944819
std::numbers::pi: 3.141592654
std::numbers::inv_pi: 0.3183098862
std::numbers::inv_sqrtpi: 0.5641895835
std::numbers::ln2: 0.6931471806
std::numbers::sqrt2: 1.414213562
std::numbers::sqrt3: 1.732050808
std::numbers::inv_sqrt3: 0.5773502692
std::numbers::egamma: 0.5772156649
std::numbers::phi: 1.618033989

C:\Users\seminar>

```

Использование математических констант

Математические константы доступны для `float`, `double` и `long double`. По умолчанию используется `double`, но вы можете указать преобразование типа к `float` (`std::numbers::pi_v<float>`) или `long double` (`std::numbers::pi_v<long double>`).

### 5.4.3 Вычисление середины отрезка и линейная интерполяция

- `std::midpoint(a,b)` вычисляет среднюю точку  $(a + (b-a)/2)$  для целых чисел, чисел с плавающей точкой и указателей. Если `a` и `b` являются указателями, то они должны указывать в один и тот же массив. Функция описана в заголовочном файле `<numeric>`.
- `std::lerp(a,b,t)` вычисляет результат линейной интерполяции  $(a + t(b - a))$ . Когда `t` вне диапазона  $[0,1]$ , то вычисляется линейная экстраполяция. Для этой функции требуется подключить заголовочный файл `<cmath>`.

Программа `midpointLerp.cpp` использует обе эти функции.

Вычисление середины отрезка и линейной интерполяции для чисел

---

```

1 // midpointLerp.cpp
2
3 #include <cmath>
4 #include <numeric>
5 #include <iostream>
6
7 int main() {
8
9     std::cout << '\n';
10

```

---

```

11  std::cout << "std::midpoint(10, 20): " << std::midpoint(10, 20) << '\n';
12
13  std::cout << '\n';
14
15  for (auto v: {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}) {
16      std::cout << "std::lerp(10, 20, " << v << "): " << std::lerp(10, 20, v)
17          << '\n';
18  }
19
20  std::cout << '\n';
21
22  }

```

---

Программа понятна без всяких объяснений.

---

```

1  // midpointlerp.cpp
2
3  #include <cmath>
4  #include <numeric>
5  #include <iostream>
6
7  int main() {
8
9      std::cout << '\n';
10
11     std::cout << "std::midpoint(10, 20): " << std::midpoint(10, 20) << '\n';
12
13     std::cout << '\n';
14
15     for (auto v: {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}) {
16         std::cout << "std::lerp(10, 20, " << v << "): " << std::lerp(10, 20, v)
17             << '\n';
18     }
19
20     std::cout << '\n';
21
22 }

```

---

Вычисление середины отрезка и линейной интерполяции для чисел

## 5.4.4 Работа с битами

Заголовочный файл `<bit>` содержит функции для доступа и манипулирования отдельными битами или последовательностями битов.

### 5.4.4.1 `std::endian`

Благодаря новому типу `std::endian` вы можете проверить порядок байтов (endianness) для заданного скалярного типа. Такой порядок может быть порядком от старшего к младшему (big-endian) и порядком от младшего к старшему (little-endian). Порядок от старшего к младшему означает, что наиболее значащий байт – это крайний левый; порядок от младшего к старшему означает, что наиболее значащий байт – это крайний справа. Под скалярным ти-

пом подразумевается арифметический тип, `enum`, указатель, указатель на поле объекта или `std::nullptr_t`.

Класс `endian` проверяет порядок байтов для всех скалярных типов.

```
enum class endian
```

---

```
enum class endian
{
    little = /*implementation-defined*/,
    big    = /*implementation-defined*/,
    native = /*implementation-defined*/
};
```

---

- Если все скалярные типы с порядком от младшего к старшему, то `std::endian::native` равно `std::endian::little`.
- Если все скалярные типы с порядком от старшего к младшему, то `std::endian::native` равно `std::endian::big`.

Поддерживаются даже пограничные случаи:

- если у всех скалярных типов размер равен 1 и порядок байтов не играет никакой роли, то значения `std::endian::native`, `std::endian::little` и `std::endian::big` совпадают.
- если платформа, на которой выполняется программа, поддерживает смешанный порядок байтов, то `std::endian::native` не равен ни `std::endian::little`, ни `std::endian::big`.

Когда я запускаю программу `getEndianness.cpp` на архитектуре `x86`, то получаю ответ `std::endian::little`.

```
enum class endian
```

---

```
// getEndianness.cpp
```

```
#include <bit>
```

```
#include <iostream>
```

```
int main() {
```

```
    if constexpr (std::endian::native == std::endian::big) {
        std::cout << "big-endian" << '\n';
    }
```

```
    else if constexpr (std::endian::native == std::endian::little) {
        std::cout << "little-endian" << '\n';    // little-endian
    }
```

```
}
```

---

Здесь используется `constexpr if` для условной компиляции кода. Это значит, что компиляция зависит от порядка байтов на архитектуре компьютера, где запущена программа.

#### 5.4.4.2 Работа с битами и битовыми последовательностями

Следующая таблица дает обзор всех функций для работы с битами. Вы можете найти их в заголовочном файле `<bit>`.

| Функция                          | Описание                                                            |
|----------------------------------|---------------------------------------------------------------------|
| <code>std::bit_cast</code>       | Выполнить интерпретацию представления объекта                       |
| <code>std::has_single_bit</code> | Проверить, является ли число степенью двух                          |
| <code>std::bit_ceil</code>       | Найти минимальную целую степень двух, не меньшую заданного числа    |
| <code>std::bit_floor</code>      | Найти максимальную целую степень двух, не большую заданного числа   |
| <code>std::bit_width</code>      | Найти наименьшее количество битов для представления данного числа   |
| <code>std::rotl</code>           | Выполнить побитовый циклический сдвиг влево                         |
| <code>std::rotr</code>           | Выполнить побитовый циклический сдвиг вправо                        |
| <code>std::countl_zero</code>    | Найти количество подряд идущих 0, начиная с наиболее значащего бита |
| <code>std::countl_one</code>     | Найти количество подряд идущих 1, начиная с наиболее значащего бита |
| <code>std::countr_zero</code>    | Найти количество подряд идущих 0, начиная с наименее значащего бита |
| <code>std::countr_one</code>     | Найти количество подряд идущих 1, начиная с наименее значащего бита |
| <code>std::popcount</code>       | Найти количество 1 в беззнаковом целом числе                        |

Все из этих функций, кроме `std::bit_cast`, требуют значения беззнакового целого типа (`unsigned char`, `unsigned short`, `unsigned int`, `unsigned long` или `unsigned long long`).

Программа `bit.cpp` демонстрирует применение этих функций.

Работа с битами

---

```
// bit.cpp
```

```
#include <bit>
#include <bitset>
#include <iostream>
```

```
int main() {

    std::uint8_t num= 0b00110010;

    std::cout << std::boolalpha;

    std::cout << "std::has_single_bit(0b00110010): " << std::has_single_bit(num)
        << '\n';

    std::cout << "std::bit_ceil(0b00110010): " << std::bitset<8>(std::bit_ceil(num))
        << '\n';
    std::cout << "std::bit_floor(0b00110010): "
        << std::bitset<8>(std::bit_floor(num)) << '\n';

    std::cout << "std::bit_width(5u): " << std::bit_width(5u) << '\n';

    std::cout << "std::rotl(0b00110010, 2): " << std::bitset<8>(std::rotl(num, 2))
        << '\n';
    std::cout << "std::rotr(0b00110010, 2): " << std::bitset<8>(std::rotr(num, 2))
        << '\n';

    std::cout << "std::countl_zero(0b00110010): " << std::countl_zero(num) << '\n';
    std::cout << "std::countl_one(0b00110010): " << std::countl_one(num) << '\n';
    std::cout << "std::countr_zero(0b00110010): " << std::countr_zero(num) << '\n';
    std::cout << "std::countr_one(0b00110010): " << std::countr_one(num) << '\n';
    std::cout << "std::popcount(0b00110010): " << std::popcount(num) << '\n';

}
```

---

Ниже приводится вывод программы.

```
std::has_single_bit(0b00110010): false
std::bit_ceil(0b00110010): 01000000
std::bit_floor(0b00110010): 00100000
std::bit_width(5u): 3
std::rotl(0b00110010, 2): 11001000
std::rotr(0b00110010, 2): 10001100
std::countl_zero(0b00110010): 2
std::countl_one(0b00110010): 0
std::countr_zero(0b00110010): 1
std::countr_one(0b00110010): 0
std::popcount(0b00110010): 3
```

Следующая программа показывает `std::bit_floor`, `std::bit_ceil`, `std::bit_width` и `std::bit_popcount` для чисел от 2 до 7.

Значения `std::bit_floor`, `std::bit_ceil`, `std::bit_width` и `std::bit_popcount` для нескольких чисел

---

```
// bitFloorCeil.cpp
```

```
#include <bit>
#include <bitset>
#include <iostream>

int main() {

    std::cout << '\n';

    std::cout << std::boolalpha;

    for (auto i = 2u; i < 8u; ++i) {
        std::cout << "bit_floor(" << std::bitset<8>(i) << ") = "
                  << std::bit_floor(i) << '\n';

        std::cout << "bit_ceil(" << std::bitset<8>(i) << ") = "
                  << std::bit_ceil(i) << '\n';

        std::cout << "bit_width(" << std::bitset<8>(i) << ") = "
                  << std::bit_width(i) << '\n';

        std::cout << "popcount(" << std::bitset<8>(i) << ") = "
                  << std::popcount(i) << '\n';

        std::cout << '\n';
    }

    std::cout << '\n';
}
```

---

```
// bitFloorCeil.cpp

#include <bits>
#include <bitset>
#include <iostream>

int main() {

    std::cout << "\n";

    std::cout << std::boolalpha;

    for (auto i = 3u; i < 8u; ++i) {
        std::cout << "bit_floor" << std::bitset<8>(i) << " = "
                  << std::bit_floor(i) << "\n";

        std::cout << "bit_ceil" << std::bitset<8>(i) << " = "
                  << std::bit_ceil(i) << "\n";

        std::cout << "bit_width" << std::bitset<8>(i) << " = "
                  << std::bit_width(i) << "\n";

        std::cout << "popcount" << std::bitset<8>(i) << " = "
                  << std::popcount(i) << "\n";

        std::cout << "\n";
    }

    std::cout << "\n";
}
```

Значения `std::bit_floor`, `std::bit_ceil`, `std::bit_width` и `std::bit_popcount` для нескольких чисел



### Краткая информация

- ♦ Функции `std::cmp_*` в C++20 поддерживают безопасное сравнение целых чисел, поскольку они поддерживают сравнение знакового и беззнакового целых чисел. В случае небезопасного сравнения возникает ошибка.
- ♦ Многие математические константы теперь определены.
- ♦ C++20 предоставляет специальные функции для вычисления среднего значения и линейной интерполяции двух значений.
- ♦ Доступны новые функции для работы с отдельными битами или последовательностями битов.



## 5.5 Календарные зоны и часовые пояса



Сиппи изучает календарь



### Отсутствие поддержки компилятора

В конце 2020 года ни один компилятор C++20 не поддерживал расширения `chrono`. Благодаря прототипу – библиотеке `date`<sup>1</sup> от Говарда Хиннанта, которая является надмножеством расширенной функциональности в C++20, – я мог поэкспериментировать. Библиотека находится на GitHub. Есть несколько способов попробовать этот прототип:

- ♦ вы можете попробовать его в Wandbox. Говард выложил заголовочный файл `date.h`, которого вполне достаточно для работы с новым типом `std::time_of_day` и календарем. Вот ссылка от Говарда: Try it on Wandbox!<sup>2</sup>;
- ♦ скопируйте заголовочный файл `date.h` в путь поиска для вашего компилятора C++;
- ♦ скачайте проект и соберите его. Уже упоминавшаяся страница `date` на GitHub<sup>3</sup> содержит дополнительную информацию. Этот шаг требуется, если вы хотите попробовать новые функции по работе с временными зонами.

Примеры в этой главе используют библиотеку Говарда Хиннанта. Но мои объяснения основаны на терминологии C++20. Когда компиляторы C++20 будут поддерживать расширенную функциональность `chrono`, я изменю примеры, чтобы они соответствовали синтаксису C++20.

<sup>1</sup> <https://github.com/HowardHinnant/date>.

<sup>2</sup> <https://wandbox.org/permlink/L8MwjzSSC3fXXrMd>.

<sup>3</sup> <https://github.com/HowardHinnant/date>.

C++20 добавляет новые компоненты в библиотеку `chrono`:

- **время дня** – это продолжительность времени с полуночи, разбитая на часы, минуты, секунды и дробные части секунд;
- **календарь** обозначает различные даты в календаре, такие как год, месяц, день, день недели или  $n$ -й день недели;
- **часовой пояс** представляет собой время, заданное для какой-то географической области.

По сути, функционал часовых поясов (C++20) основан на функционале календаря (C++20), а функционал календаря (C++20) основан на библиотеке `chrono` (C++11).



### Библиотека работы со временем в C++11

Для того чтобы получить наибольшее количество информации из этой главы, важно базовое понимание библиотеки `chrono`. В C++11 имеется три главных компонента для работы со временем:

- ♦ **точка во времени** (time point) определяется стартовым временем, так называемым `epoch` и дополнительной продолжительностью (duration);
- ♦ **продолжительность** (интервал) – это разница во времени между двумя точками во времени. Определяется числом отсчетов (ticks);
- ♦ **часы** – состоят из стартовой точки и отсчета, так что может быть вычислена текущая точка во времени.

Честно говоря, все это довольно сложно. С одной стороны, у каждого есть свое интуитивное понятие времени, но с другой – крайне сложно дать ему формальное определение. Например, три компонента – точка во времени, продолжительность и часы – зависят друг от друга. Если вы хотите узнать больше о функциональности, связанной со временем в C++11, то почитайте мой пост<sup>1</sup> об этом.

Но это еще не все. C++20 вводит несколько новых часов. Благодаря библиотеке форматирования в C++20 интервалы могут быть удобно записаны или прочитаны.

## 5.5.1 Время дня

`std::chrono::hh_mm_ss` – это время начиная с полуночи, разбитое на часы, минуты, секунды и дробные части секунд. Этот тип обычно используется при форматировании. Следующая таблица дает вам краткий обзор `tofDay` – экземпляра класса `std::chrono::hh_mm_ss`.

| Функция                       | Описание                                                  |
|-------------------------------|-----------------------------------------------------------|
| <code>tofDay.hours()</code>   | Возвращает количество часов с полуночи (компоненту часов) |
| <code>tofDay.minutes()</code> | Возвращает количество минут, прошедших с полуночи         |

<sup>1</sup> <https://www.modernescpp.com/index.php/tag/time>.

Окончание табл.

| Функция                                | Описание                                                                            |
|----------------------------------------|-------------------------------------------------------------------------------------|
| <code>timeOfDay.seconds()</code>       | Возвращает количество секунд для времени, прошедшего с полуночи                     |
| <code>timeOfDay.subseconds()</code>    | Возвращает количество дробного количества секунд для времени, прошедшего с полуночи |
| <code>timeOfDay.to_duration()</code>   | Возвращает продолжительность времени с полуночи                                     |
| <code>std::chrono::make12(hour)</code> | Возвращает 12-часовой эквивалент для 24-часового времени                            |
| <code>std::chrono::make24(hour)</code> | Возвращает 24-часовой эквивалент для 12-часового времени                            |
| <code>std::chrono::is_am(hour)</code>  | Проверяет, является ли заданное 24-часовое время AM                                 |
| <code>std::chrono::is_pm(hour)</code>  | Проверяет, является ли заданное 24-часовое время PM                                 |

Использование этих функций довольно просто.

Время дня

```

1 // timeOfDay.cpp
2
3 #include "date.h"
4 #include <iostream>
5
6 int main() {
7     using namespace date;
8
9     using namespace std::chrono_literals;
10
11     std::cout << std::boolalpha << '\n';
12     auto timeOfDay = date::hh_mm_ss(10.5h + 98min + 2020s + 0.5s);
13
14     std::cout << "timeOfDay: " << timeOfDay << '\n';
15
16     std::cout << '\n';
17
18     std::cout << "timeOfDay.hours(): " << timeOfDay.hours() << '\n';
19     std::cout << "timeOfDay.minutes(): " << timeOfDay.minutes() << '\n';
20     std::cout << "timeOfDay.seconds(): " << timeOfDay.seconds() << '\n';
21     std::cout << "timeOfDay.subseconds(): " << timeOfDay.subseconds() << '\n';
22     std::cout << "timeOfDay.to_duration(): " << timeOfDay.to_duration() << '\n';
23
24     std::cout << '\n';
25
26     std::cout << "date::hh_mm_ss(45700.5s): " << date::hh_mm_ss(45700.5s) << '\n';
27

```

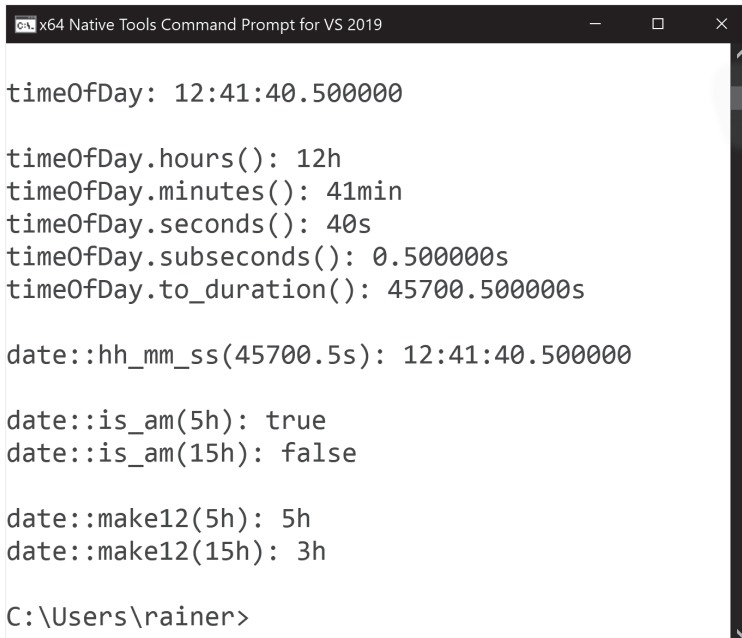
```

28  std::cout << '\n';
29
30  std::cout << "date::is_am(5h): " << date::is_am(5h) << '\n';
31  std::cout << "date::is_am(15h): " << date::is_am(15h) << '\n';
32
33  std::cout << '\n';
34
35  std::cout << "date::make12(5h): " << date::make12(5h) << '\n';
36  std::cout << "date::make12(15h): " << date::make12(15h) << '\n';
37
38  }

```

Вначале в строке 12 создается экземпляр класса `std::chrono::hh_mm_ss timeOfDay`. Благодаря литералам времени, появившимся в стандарте C++14, я легко могу задать несколько продолжительностей времени, чтобы инициализировать объект `timeOfDay`. В C++20 вы можете непосредственно выводить `timeOfDay` (строка 14). Это та самая причина, по которой я использую пространство имен `date` в строке 7. Оставшаяся часть кода достаточно простая и ясная для понимания. Строки 18–21 выводят компоненты времени с полуночи в часах, минутах, секундах и долях секунды. Строка 22 возвращает продолжительность времени с полуночи в секундах. Строка 26 более интересна: заданное количество секунд соответствует времени в строке 15. Строки 30 и 32 выводят, является ли время АМ (до полудня). Строки 35 и 36 выводят 12-часовой эквивалент заданного времени в часах.

Вывод программы:



```

timeOfDay: 12:41:40.500000

timeOfDay.hours(): 12h
timeOfDay.minutes(): 41min
timeOfDay.seconds(): 40s
timeOfDay.subseconds(): 0.500000s
timeOfDay.to_duration(): 45700.500000s

date::hh_mm_ss(45700.5s): 12:41:40.500000

date::is_am(5h): true
date::is_am(15h): false

date::make12(5h): 5h
date::make12(15h): 3h

C:\Users\rainer>

```

## 5.5.2 Календарные даты

Новый тип, появившийся в C++20 в библиотеке `chrono`, – это календарная дата. C++20 поддерживает различные способы задания календарных дат и взаимодействия с ними. Но прежде всего давайте разберемся с тем, что такое календарная дата.

- **Календарная дата** – это дата (день), состоящая из года, месяца и дня. Соответственно, в C++20 есть тип `std::chrono::year_month_day`. Но в C++20 есть не только это. Следующая таблица дает первоначальный обзор типов, связанных с датами, после чего я перейду к различным примерам использования.

Различные типы календарных дат

| Тип                                                   | Описание                                                   |
|-------------------------------------------------------|------------------------------------------------------------|
| <code>std::chrono::last_spec</code>                   | Обозначает последний день или день недели месяца           |
| <code>std::chrono::day</code>                         | Представляет день месяца                                   |
| <code>std::chrono::month</code>                       | Представляет месяц года                                    |
| <code>std::chrono::year</code>                        | Представляет год в григорианском календаре                 |
| <code>std::chrono::weekday</code>                     | Представляет день недели в григорианском календаре         |
| <code>std::chrono::weekday_indexed</code>             | Представляет $n$ -й день недели месяца                     |
| <code>std::chrono::weekday_last</code>                | Представляет последний день недели месяца                  |
| <code>std::chrono::month_day</code>                   | Представляет конкретный день конкретного месяца            |
| <code>std::chrono::month_day_last</code>              | Представляет последний день заданного месяца               |
| <code>std::chrono::month_weekday</code>               | Представляет $n$ -й день недели месяца                     |
| <code>std::chrono::month_weekday_last</code>          | Представляет последний день недели заданного месяца        |
| <code>std::chrono::year_month</code>                  | Представляет заданный месяц заданного года                 |
| <code>std::chrono::year_month_day</code>              | Представляет заданный год, месяц и день                    |
| <code>std::chrono::year_month_day_last</code>         | Представляет последний день заданного месяца и года        |
| <code>std::chrono::year_month_weekday</code>          | Представляет $n$ -й день недели заданного года и месяца    |
| <code>std::chrono::year_month_day_weekday_last</code> | Представляет последний день недели заданного года и месяца |
| <code>std::chrono::operator /</code>                  | Создает дату в григорианском календаре                     |

Давайте начнем с простого примера и создадим несколько дат.

### 5.5.2.1 Создание календарных дат

Программа `createCalendar.cpp` показывает различные способы создания календарных дат.

Создание календарных дат

---

```
1 // createCalendar.cpp
2
3 #include <iostream>
4 #include "date.h"
5
6 int main() {
7
8     std::cout << '\n';
9
10    using namespace date;
11
12    constexpr auto yearMonthDay{year(1940)/month(6)/day(26)};
13    std::cout << yearMonthDay << " ";
14    std::cout << date::year_month_day(1940_y, June, 26_d) << '\n';
15
16    std::cout << '\n';
17
18    constexpr auto yearMonthDayLast{year(2010)/March/last};
19    std::cout << yearMonthDayLast << " ";
20    std::cout << date::year_month_day_last(2010_y, month_day_last(month(3))) << '\n';
21
22    constexpr auto yearMonthWeekday{year(2020)/March/Thursday[2]};
23    std::cout << yearMonthWeekday << " ";
24    std::cout << date::year_month_weekday(2020_y, month(March), Thursday[2]) << '\n';
25
26    constexpr auto yearMonthWeekdayLast{year(2010)/March/Monday[last]};
27    std::cout << yearMonthWeekdayLast << " ";
28    std::cout << date::year_month_weekday_last(2010_y, month(March),
29                                              weekday_last(Monday)) << '\n';
30
31    std::cout << '\n';
32
33    constexpr auto day_{day(19)};
34    std::cout << day_ << " ";
35    std::cout << date::day(19) << '\n';
36
37    constexpr auto month_{month(1)};
38    std::cout << month_ << " ";
39    std::cout << date::month(1) << '\n';
40
41    constexpr auto year_{year(1988)};
42    std::cout << year_ << " ";
```

```

43 std::cout << date::year(1988) << '\n';
44
45 constexpr auto weekday_{weekday(5)};
46 std::cout << weekday_ << " ";
47 std::cout << date::weekday(5) << '\n';
48
49 constexpr auto yearMonth{year(1988)/1};
50 std::cout << yearMonth << " ";
51 std::cout << date::year_month(year(1988), January) << '\n';
52
53 constexpr auto monthDay{10/day(22)};
54 std::cout << monthDay << " ";
55 std::cout << date::month_day(October, day(22)) << '\n';
56
57 constexpr auto monthDayLast{June/last};
58 std::cout << monthDayLast << " ";
59 std::cout << date::month_day_last(month(6)) << '\n';
60
61 constexpr auto monthWeekday{2/Monday[3]};
62 std::cout << monthWeekday << " ";
63 std::cout << date::month_weekday(February, Monday[3]) << '\n';
64
65 constexpr auto monthWeekDayLast{June/Sunday[last]};
66 std::cout << monthWeekDayLast << " ";
67 std::cout << date::month_weekday_last(June, weekday_last(Sunday)) << '\n';
68
69 std::cout << '\n';
70
71 }

```

Есть два разных способа создания календарных дат. Вы можете использовать специальный синтаксис (*cute syntax*) `yearMonthDay{year(1940)/month(6)/day(26)}` (строка 12) или явно заданный тип `date::year_month_day(1940y, June, 26d)` (строка 14). Чтобы не перегружать вас, я отложу объяснение специального синтаксиса до следующего раздела. Явный тип достаточно интересен, поскольку он использует литералы даты и времени вроде `1940y`, `26d` и предопределенную константу `June`. Это была наиболее очевидная часть программы.

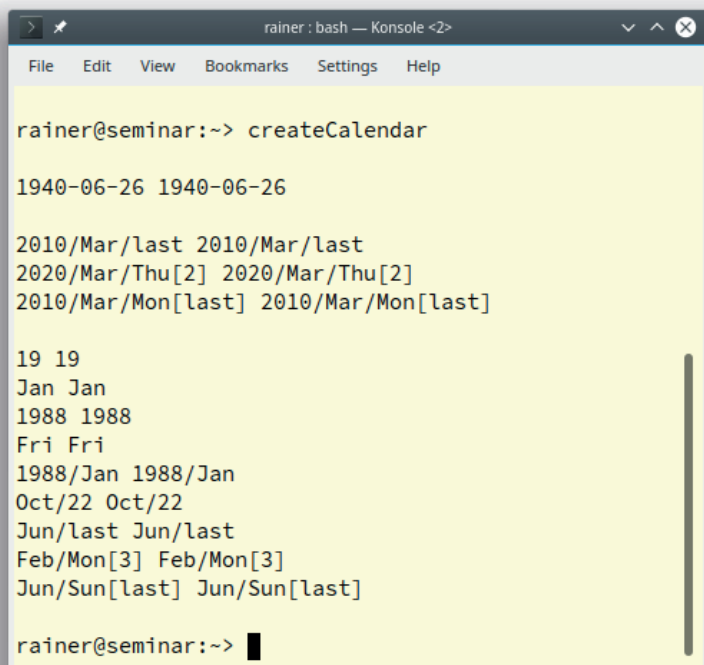
Строки 18, 22 и 26 предлагают другие способы задания дат.

- Строка 18: последний день марта 2010 года `{year(2010)/March/last}` или `year_month_day_last(2010y, month_day_last(month(3)))`.
- Строка 22: второй четверг марта 2020 года: `{year(2020)/March/Thursday[2]}` или `year_-month_weekday(2020y, month(March), Thursday[2])`.
- Строка 26: последний понедельник марта 2010 года: `{year(2010)/March/Monday[last]}` или `year_month_-weekday_last(2010y, month(March), weekday_last(Monday))`.

Оставшиеся календарные типы обозначают день (строка 33), месяц (строка 37) или год (строка 41). Вы можете совмещать их и использовать как

строительные блоки для полного задания дат, например как в строках 18, 22 или 26.

Вывод программы:



```
rainer : bash — Konsole <2>
File Edit View Bookmarks Settings Help

rainer@seminar:~> createCalendar

1940-06-26 1940-06-26

2010/Mar/last 2010/Mar/last
2020/Mar/Thu[2] 2020/Mar/Thu[2]
2010/Mar/Mon[last] 2010/Mar/Mon[last]

19 19
Jan Jan
1988 1988
Fri Fri
1988/Jan 1988/Jan
Oct/22 Oct/22
Jun/last Jun/last
Feb/Mon[3] Feb/Mon[3]
Jun/Sun[last] Jun/Sun[last]

rainer@seminar:~> █
```

Различные календарные даты

Как и обещал, сейчас я расскажу про специальный синтаксис (cute syntax) для дат.

### 5.5.2.2 Специальный синтаксис

Этот синтаксис состоит из переопределенного оператора деления, используемого для задания даты. Перегруженный оператор поддерживает литералы времени (например, 2020y, 31d) и константы (January, February, March, April, May, June, July, August, September, October, November, December).

Возможны следующие три комбинации года, месяца и дня при использовании этого синтаксиса.

Специальный синтаксис



Эти комбинации выбраны не произвольно. Это используемые по всему миру комбинации. Все остальные комбинации не допустимы.



Соответственно, когда вы выбираете тип `year`, `month` или `day` для своего первого аргумента, типы для двух оставшихся уже не важны и надо указывать просто числа.

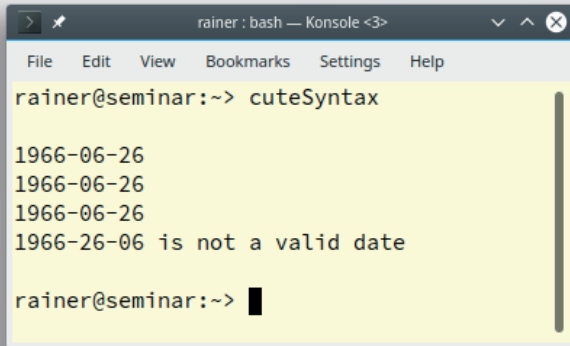
Специальный синтаксис

---

```
1 // cuteSyntax.cpp
2
3 #include <iostream>
4 #include "date.h"
5
6 int main() {
7
8     std::cout << '\n';
9
10    using namespace date;
11
12    constexpr auto yearMonthDay{year(1966)/6/26};
13    std::cout << yearMonthDay << '\n';
14
15    constexpr auto dayMonthYear{day(26)/6/1966};
16    std::cout << dayMonthYear << '\n';
17
18    constexpr auto monthDayYear{month(6)/26/1966};
19    std::cout << monthDayYear << '\n';
20
21    constexpr auto yearDayMonth{year(1966)/month(26)/6};
22    std::cout << yearDayMonth << '\n';
23
24    std::cout << '\n';
25
26 }
```

---

Комбинация год/день/месяц (строка 21) недопустима и вызывает ошибку компиляции.



```

rainer@seminar:~> cuteSyntax

1966-06-26
1966-06-26
1966-06-26
1966-26-06 is not a valid date

rainer@seminar:~>

```

Использование специального синтаксиса

Я полагаю, что вы хотите показывать календарные даты `{year(2010)/March/last}` в удобочитаемом виде, например 20-03-2010. Для этого используется оператор `local_days` или `sys_days`.

### 5.5.2.3 Показ календарных дат

Благодаря `std::chrono::local_days` или `std::chrono::sys_days` вы можете преобразовывать календарные даты в `std::chrono::time_point`. Я использую `std::chrono::sys_days` в своем примере. `std::chrono::sys_days` основан на `std::chrono::system_clock`<sup>1</sup>. Давайте переведем календарные даты (строки 18, 22 и 26) из предыдущей программы `createCalendar.cpp`.

Показ календарных дат

---

```

1  // sysDays.cpp
2
3  #include <iostream>
4  #include "date.h"
5
6  int main() {
7
8      std::cout << '\n';
9
10     using namespace date;
11
12     constexpr auto yearMonthDayLast{year(2010)/March/last};
13     std::cout << "sys_days(yearMonthDayLast): "
14               << sys_days(yearMonthDayLast) << '\n';
15

```

<sup>1</sup> [https://en.cppreference.com/w/cpp/chrono/system\\_clock](https://en.cppreference.com/w/cpp/chrono/system_clock).

```

16 constexpr auto yearMonthWeekday{year(2020)/March/Thursday[2]};
17 std::cout << "sys_days(yearMonthWeekday): "
18           << sys_days(yearMonthWeekday) << '\n';
19
20 constexpr auto yearMonthWeekdayLast{year(2010)/March/Monday[last]};
21 std::cout << "sys_days(yearMonthWeekdayLast): "
22           << sys_days(yearMonthWeekdayLast) << '\n';
23
24 std::cout << '\n';
25
26 constexpr auto leapDate{year(2012)/February/last};
27 std::cout << "sys_days(leapDate): " << sys_days(leapDate) << '\n';
28
29 constexpr auto noLeapDate{year(2013)/February/last};
30 std::cout << "sys_day(noLeapDate): " << sys_days(noLeapDate) << '\n';
31
32 std::cout << '\n';
33
34 }

```

Константа `std::chrono::last` (строка 11) позволяет мне легко определить, сколько дней в месяце. Вывод программы показывает, что 2012 – это високосный год (строка 26), а 2013 – невисокосный (строка 29).

```

1 // sysdays.cpp
2
3 #include <iostream>
4 #include "date.h"
5
6 int main() {
7
8     std::cout << '\n';
9
10    using namespace date;
11
12    constexpr auto yearMonthDayLast{year(2010)/March/last};
13    std::cout << "sys_days(yearMonthDayLast): "
14            << sys_days(yearMonthDayLast) << '\n';
15
16    constexpr auto yearMonthWeekday{year(2020)/March/Thursday[2]};
17    std::cout << "sys_days(yearMonthWeekday): "
18            << sys_days(yearMonthWeekday) << '\n';
19

```

Показ календарных дат

Пусть у вас есть календарная дата, например `year(2100)/2/29`. Вашим первым вопросом может быть: корректна ли эта дата?

### 5.5.2.4 Проверка даты на корректность

Различные календарные типы в C++20 имеют встроенный метод `ok`. Этот метод возвращает `true`, если дата корректна.

Проверка даты на корректность

---

```
1  // leapYear.cpp
2
3  #include <iostream>
4  #include "date.h"
5
6  int main() {
7
8      std::cout << std::boolalpha << '\n';
9
10     using namespace date;
11
12     std::cout << "Valid days" << '\n';
13     day day31(31);
14     day day32 = day31 + days(1);
15     std::cout << "  day31: " << day31 << "; ";
16     std::cout << "day31.ok(): " << day31.ok() << '\n';
17     std::cout << "  day32: " << day32 << "; ";
18     std::cout << "day32.ok(): " << day32.ok() << '\n';
19
20
21     std::cout << '\n';
22
23     std::cout << "Valid months" << '\n';
24     month month1(1);
25     month month0(0);
26     std::cout << "  month1: " << month1 << "; ";
27     std::cout << "month1.ok(): " << month1.ok() << '\n';
28     std::cout << "  month0: " << month0 << "; ";
29     std::cout << "month0.ok(): " << month0.ok() << '\n';
30
31     std::cout << '\n';
32
33     std::cout << "Valid years" << '\n';
34     year year2020(2020);
35     year year32768(-32768);
36     std::cout << "  year2020: " << year2020 << "; ";
```

```

37  std::cout << "year2020.ok(): " << year2020.ok() << '\n';
38  std::cout << "  year32768: " << year32768 << "; ";
39  std::cout << "year32768.ok(): " << year32768.ok() << '\n';
40
41  std::cout << '\n';
42
43  std::cout << "Leap Years" << '\n';
44
45  constexpr auto leapYear2016{year(2016)/2/29};
46  constexpr auto leapYear2020{year(2020)/2/29};
47  constexpr auto leapYear2024{year(2024)/2/29};
48
49  std::cout << "  leapYear2016.ok(): " << leapYear2016.ok() << '\n';
50  std::cout << "  leapYear2020.ok(): " << leapYear2020.ok() << '\n';
51  std::cout << "  leapYear2024.ok(): " << leapYear2024.ok() << '\n';
52
53  std::cout << '\n';
54
55  std::cout << "No Leap Years" << '\n';
56
57  constexpr auto leapYear2100{year(2100)/2/29};
58  constexpr auto leapYear2200{year(2200)/2/29};
59  constexpr auto leapYear2300{year(2300)/2/29};
60
61  std::cout << "  leapYear2100.ok(): " << leapYear2100.ok() << '\n';
62  std::cout << "  leapYear2200.ok(): " << leapYear2200.ok() << '\n';
63  std::cout << "  leapYear2300.ok(): " << leapYear2300.ok() << '\n';
64
65  std::cout << '\n';
66
67  std::cout << "Leap Years" << '\n';
68
69  constexpr auto leapYear2000{year(2000)/2/29};
70  constexpr auto leapYear2400{year(2400)/2/29};
71  constexpr auto leapYear2800{year(2800)/2/29};
72
73  std::cout << "  leapYear2000.ok(): " << leapYear2000.ok() << '\n';
74  std::cout << "  leapYear2400.ok(): " << leapYear2400.ok() << '\n';
75  std::cout << "  leapYear2800.ok(): " << leapYear2800.ok() << '\n';
76

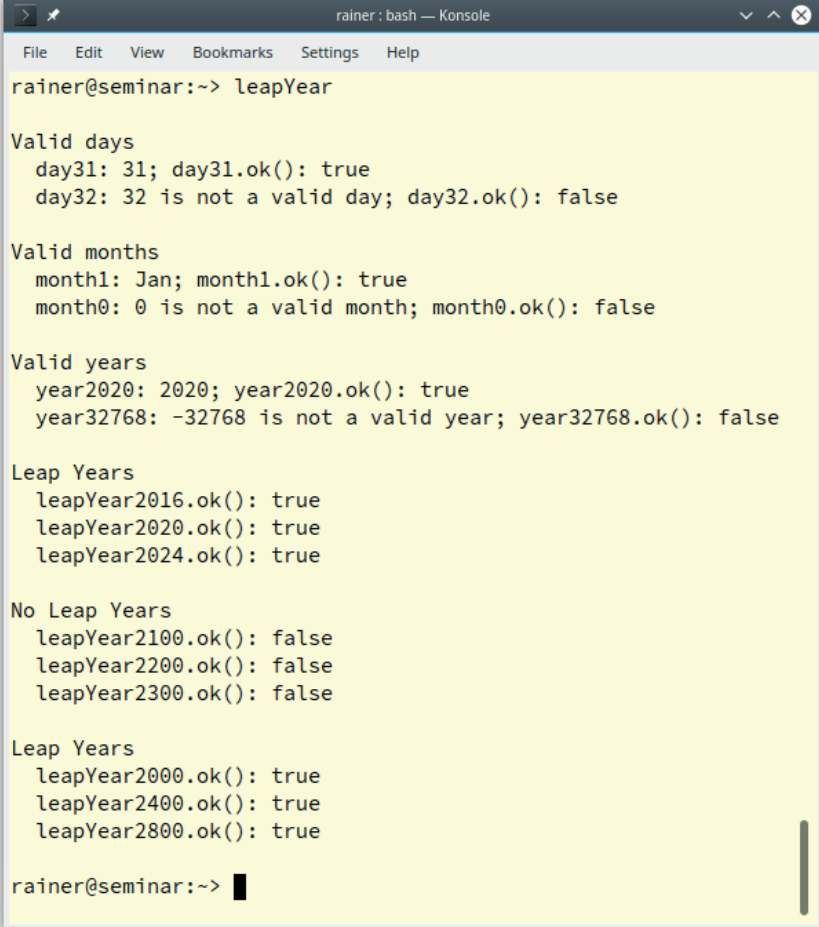
```

```

77  std::cout << '\n';
78
79  }

```

Я проверяю в программе, является ли заданный день (строка 12), заданный месяц (строка 25) или заданный год (строка 33) корректным. Диапазон для дня [1,31], для месяца [1,12] и для года [-32767, 32767]. Соответственно, функция `ok` для всех этих значений возвращает `false`. Есть два интересных момента, связанных с выводом различных значений. Во-первых, если значение не действительно, то при выводе мы получаем «is not a valid day», «is not a valid month» или «is not a valid year». Во-вторых, месяцы выводятся в текстовом представлении.



```

rainer@seminar:~$ ./leapYear

Valid days
  day31: 31; day31.ok(): true
  day32: 32 is not a valid day; day32.ok(): false

Valid months
  month1: Jan; month1.ok(): true
  month0: 0 is not a valid month; month0.ok(): false

Valid years
  year2020: 2020; year2020.ok(): true
  year32768: -32768 is not a valid year; year32768.ok(): false

Leap Years
  leapYear2016.ok(): true
  leapYear2020.ok(): true
  leapYear2024.ok(): true

No Leap Years
  leapYear2100.ok(): false
  leapYear2200.ok(): false
  leapYear2300.ok(): false

Leap Years
  leapYear2000.ok(): true
  leapYear2400.ok(): true
  leapYear2800.ok(): true

rainer@seminar:~$

```

Проверка даты на корректность

Вы можете легко применить вызов `ok` к календарной дате. Теперь легко проверить, является ли заданный день високосным и является ли соответствующей

щий год високосным. В распространенном григорианском календаре<sup>1</sup> применяются следующие правила.

Каждый год, который делится на 4, является **високосным годом**.

- За исключением годов, которые кратны 100. Они не високосные:
  - ◆ за исключением годов, которые кратны 400, они високосные.

Слишком сложно? Программа `leapYears.cpp` демонстрирует это правило.

Расширенная библиотека `chrono` делает очень легким расчет разницы между двумя календарными датами.

### 5.5.2.5 Арифметика с календарными датами

Следующая программа `queryCalendarDates.cpp` запрашивает несколько дат.

Арифметика с календарными датами

---

```

1 // queryCalendarDates.cpp
2
3 #include "date.h"
4 #include <iostream>
5
6 int main() {
7
8     using namespace date;
9
10    std::cout << '\n';
11
12    auto now = std::chrono::system_clock::now();
13    std::cout << "The current time is: " << now << " UTC\n";
14    std::cout << "The current date is: " << floor<days>(now) << '\n';
15    std::cout << "The current date is: " << year_month_day{floor<days>(now)}
16              << '\n';
17    std::cout << "The current date is: " << year_month_weekday{floor<days>(now)}
18              << '\n';
19
20    std::cout << '\n';
21
22
23    auto currentDate = year_month_day(floor<days>(now));
24    auto currentYear = currentDate.year();
25    std::cout << "The current year is " << currentYear << '\n';
26    auto currentMonth = currentDate.month();
27    std::cout << "The current month is " << currentMonth << '\n';
28    auto currentDay = currentDate.day();
29    std::cout << "The current day is " << currentDay << '\n';
30
31    std::cout << '\n';
32

```

<sup>1</sup> [https://en.wikipedia.org/wiki/Gregorian\\_calendar](https://en.wikipedia.org/wiki/Gregorian_calendar).

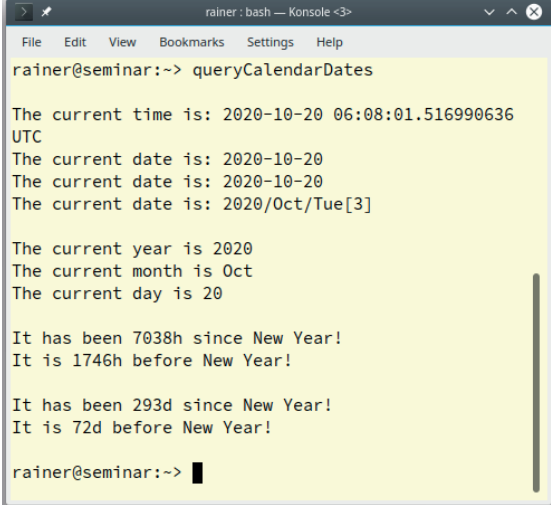
```

33  auto hAfter = floor<std::chrono::hours>(now) - sys_days(January/1/currentYear);
34  std::cout << "It has been " << hAfter << " since New Year!\n";
35  auto nextYear = currentDate.year() + years(1);
36  auto nextNewYear = sys_days(January/1/nextYear);
37  auto hBefore = sys_days(January/1/nextYear) - floor<std::chrono::hours>(now);
38  std::cout << "It is " << hBefore << " before New Year!\n";
39
40  std::cout << '\n';
41
42  std::cout << "It has been " << floor<days>(hAfter) << " since New Year!\n";
43  std::cout << "It is " << floor<days>(hBefore) << " before New Year!\n";
44
45  std::cout << '\n';
46
47  }

```

В стандарте C++20 вы можете непосредственно вывести точку во времени, например `now` (строка 12). `std::chrono::floor` переводит точку во времени в дни `std::chrono::sys_days`. Это значение может быть использовано для инициализации календарного типа `std::chrono::year_month_day`. Наконец, когда я помещаю значение в `std::chrono::year_month_weekday`, то я получаю ответ, что этот день – это 3-й вторник в октябре. Конечно, я также могу получить компоненты календарной даты, такие как год, месяц и число (строка 23).

Наиболее интересной здесь является строка 33. Когда я вычитаю из текущей даты, используя разрешение на уровне часов, первое января текущего года, то я получаю количество часов начиная с нового года. Когда я вычитаю из первого января следующего года (строка 37) текущую дату, работая на уровне часов, то я получаю количество часов до нового года. Но, может быть, мне не нравится работать с часами. Строки 42 и 43 показывают эти значения, работая на уровне дней.



```

rainer@seminar:~$ queryCalendarDates

The current time is: 2020-10-20 06:08:01.516990636
UTC
The current date is: 2020-10-20
The current date is: 2020-10-20
The current date is: 2020/Oct/Tue[3]

The current year is 2020
The current month is Oct
The current day is 20

It has been 7038h since New Year!
It is 1746h before New Year!

It has been 293d since New Year!
It is 72d before New Year!

rainer@seminar:~$

```

Арифметика с календарными датами



Теперь я хочу узнать день недели моего дня рождения.

### 5.5.2.6 Работаем с днями недели

Благодаря расширенной библиотеке chrono стало очень легко получить день недели для любого заданного дня.

День недели по календарной дате

---

```

1 // weekdaysOfBirthdays.cpp
2
3 #include <cstdlib>
4 #include <iostream>
5 #include "date.h"
6
7 int main() {
8
9     std::cout << '\n';
10
11     using namespace date;
12
13     int y;
14     int m;
15     int d;
16
17     std::cout << "Year: ";
18     std::cin >> y;
19     std::cout << "Month: ";
20     std::cin >> m;
21     std::cout << "Day: ";
22     std::cin >> d;
23
24     std::cout << '\n';
25
26     auto birthday = year(y)/month(m)/day(d);
27
28     if (not birthday.ok()) {
29         std::cout << birthday << '\n';
30         std::exit(EXIT_FAILURE);
31     }
32
33     std::cout << "Birthday: " << birthday << '\n';
34     auto birthdayWeekday = year_month_weekday(birthday);
35     std::cout << "Weekday of birthday: " << birthdayWeekday.weekday() << '\n';
36

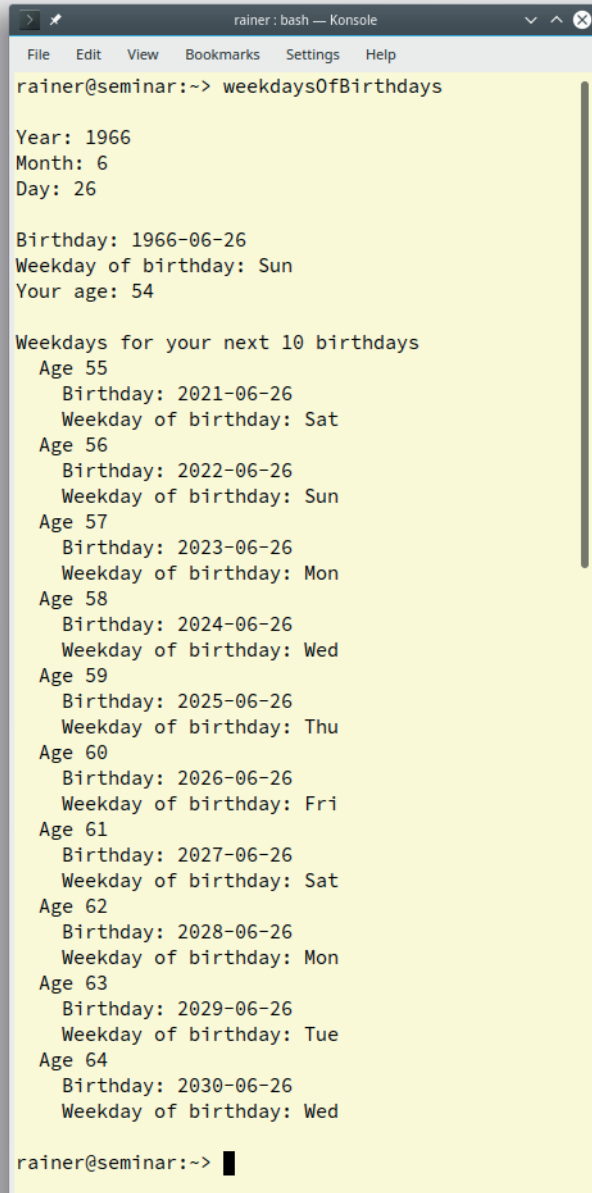
```

```
37 auto currentDate = year_month_day(floor<days>(
38                                     std::chrono::system_clock::now()));
39 auto currentYear = currentDate.year();
40
41 auto age = (int)currentDate.year() - (int)birthday.year();
42 std::cout << "Your age: " << age << '\n';
43
44 std::cout << '\n';
45
46 std::cout << "Weekdays for your next 10 birthdays" << '\n';
47
48 for (int i = 1, newYear = (int)currentYear; i <= 10; ++i) {
49     std::cout << " Age " << ++age << '\n';
50     auto newBirthday = year(++newYear)/month(m)/day(d);
51     std::cout << " Birthday: " << newBirthday << '\n';
52     std::cout << " Weekday of birthday: "
53                 << year_month_weekday(newBirthday).weekday() << '\n';
54 }
55
56 std::cout << '\n';
57
58 }
```

---

Сначала программа спрашивает у вас год, день и месяц вашего рождения (строка 17). Исходя из введенных данных, строится календарная дата (строка 26), после чего она проверяется на корректность (строка 28). Теперь я выведу день недели для вашего дня рождения. Я использую календарную дату для задания значения `std::chrono::year_month_weekday` (строка 34). Для получения целочисленного представления календарного типа `year` я преобразую его в `int` (строка 41). Теперь могу вывести ваш возраст. Наконец, в цикле выводится для каждого из ваших следующих 10 дней рождения следующая информация: ваш возраст, календарная дата и соответствующий ей день недели. Для этого мне нужно просто увеличить значение переменных `age` и `newYear` соответственно.

Вывод программы приведен ниже.



```
rainer: bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~> weekdaysOfBirthdays

Year: 1966
Month: 6
Day: 26

Birthday: 1966-06-26
Weekday of birthday: Sun
Your age: 54

Weekdays for your next 10 birthdays
Age 55
  Birthday: 2021-06-26
  Weekday of birthday: Sat
Age 56
  Birthday: 2022-06-26
  Weekday of birthday: Sun
Age 57
  Birthday: 2023-06-26
  Weekday of birthday: Mon
Age 58
  Birthday: 2024-06-26
  Weekday of birthday: Wed
Age 59
  Birthday: 2025-06-26
  Weekday of birthday: Thu
Age 60
  Birthday: 2026-06-26
  Weekday of birthday: Fri
Age 61
  Birthday: 2027-06-26
  Weekday of birthday: Sat
Age 62
  Birthday: 2028-06-26
  Weekday of birthday: Mon
Age 63
  Birthday: 2029-06-26
  Weekday of birthday: Tue
Age 64
  Birthday: 2030-06-26
  Weekday of birthday: Wed

rainer@seminar:~> █
```

### 5.5.2.7 Вычисление порядковых дат

В качестве последнего примера новой функциональности я хочу привести онлайн-ресурс Examples and Recipes<sup>1</sup> от Говарда Хинанта, у которого есть около 40 примеров новой функциональности библиотеки chrono. Наверное, расширение chrono в C++20 не так просто для понимания, поэтому так важно иметь много примеров. Вам будет полезно использовать эти примеры как точку отсчета для дальнейших экспериментов и улучшения вашего понимания этой библиотеки.

Для того чтобы получить представление об Examples and Recipes, я хочу показать программу, написанную Роландом Боком<sup>2</sup>, которая вычисляет порядковые даты.

*«Порядковая дата состоит из года и дня года (1 января – это день 1, 31 декабря – это день 365 или 366). Год может быть получен непосредственно из year\_month\_day. А вычисление дня оказывается очень простым. В коде, приведенном ниже, мы используем тот факт, что year\_month\_day может работать с неверными датами вроде 0 января» (Роланд Бок).*

Я добавил необходимые заголовочные файлы к примеру от Роланда.

Вычисление порядковых дат

---

```

1  // ordinalDate.cpp
2
3  #include "date.h"
4  #include <iomanip>
5  #include <iostream>
6
7  int main()
8  {
9      using namespace date;
10
11     const auto time = std::chrono::system_clock::now();
12     const auto daypoint = floor<days>(time);
13     const auto ymd = year_month_day{daypoint};
14
15     // calculating the year and the day of the year
16     const auto year = ymd.year();
17     const auto year_day = daypoint - sys_days{year/January/0};
18
19     std::cout << year << '-' << std::setfill('0') << std::setw(3)
20               << year_day.count() << '\n';
21
```

<sup>1</sup> <https://github.com/HowardHinnant/date/wiki/Examples-and-Recipes>.

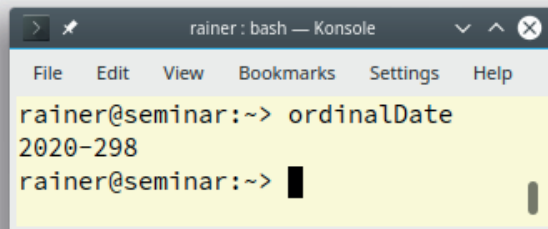
<sup>2</sup> <https://github.com/rbock>.

```

22  // inverse calculation and check
23  assert(ymd == year_month_day{sys_days{year/January/0} + year_day});
24  }

```

Сделаем несколько замечаний по этой программе. Строка 12 отсекает текущий момент времени (переводит в дни). Это значение используется в следующей строке для инициализации календарной даты. Строка 17 вычисляет продолжительность времени между двумя моментами времени. Оба этих момента имеют разрешение на уровне дней. И наконец, `year_day.count()` в строке 19 возвращает продолжительность в днях.



Вычисление порядковых дат

### 5.5.3 Часовые пояса

Прежде всего часовой пояс (time zone) – это область, содержащая такие данные, как дату, летний/зимний переносы времени и т. д. Библиотека для работы с часовыми поясами в C++20 содержит полный парсер базы по часовым поясам IANA (IANA timezone database)<sup>1</sup>. Следующая таблица даст вам общее представление о новой функциональности библиотеки.

Типы, связанные с часовыми поясами

| Тип                                                                                                                                                                | Описание                                              |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------|
| <code>std::chrono::tzdb</code>                                                                                                                                     | Описывает копию базы IANA                             |
| <code>std::chrono::tdzb_list</code>                                                                                                                                | Представляет собой связанный список <code>tzdb</code> |
| <code>std::chrono::get_tzdb</code><br><code>std::chrono::get_tzdb_list</code><br><code>std::chrono::reload_tzdb</code><br><code>std::chrono::remote_version</code> | Доступ и управление глобальной базой часовых поясов   |
| <code>std::chrono::locate_zone</code>                                                                                                                              | Находит часовой пояс по имени                         |
| <code>std::chrono::current_zone</code>                                                                                                                             | Возвращает текущий часовой пояс                       |
| <code>std::chrono::time_zone</code>                                                                                                                                | Представляет часовой пояс                             |

<sup>1</sup> <https://www.iana.org/timezones>.

| Тип                                              | Описание                                                            |
|--------------------------------------------------|---------------------------------------------------------------------|
| <code>std::chrono::sys_info</code>               | Представляет информацию о часовом поясе в заданный момент времени   |
| <code>std::chrono::local_info</code>             | Представляет информацию о переводе локального времени в UNIX        |
| <code>std::chrono::zoned_traits</code>           | Класс для указателей на часовые пояса                               |
| <code>std::chrono::zoned_time</code>             | Представляет часовой пояс и момент времени                          |
| <code>std::chrono::leap_second</code>            | Содержит информацию о вставке високосной секунды                    |
| <code>std::chrono::time_zone_link</code>         | Представляет альтернативное имя для часового пояса                  |
| <code>std::chrono::nonexistent_local_time</code> | Исключение, которое срабатывает, если локальное время не существует |



### Компиляция примеров

Перед тем как я покажу вам следующие два примера, я хотел бы сделать одно замечание. Для компиляции программы, использующей библиотеку для работы с часовыми поясами, вам нужно откомпилировать файл `tz.cpp` из библиотеки `date`<sup>1</sup> и слинковать его вместе с библиотекой `curl`<sup>2</sup>. Библиотека `curl` необходима для того, чтобы скачать базу LANA<sup>3</sup>. Следующая команда `g++` даст вам представление об этом.

Компиляция с прототипов библиотеки `date`

---

```
g++ localTime.cpp -I <Path to data/tz.h> tz.cpp -std=c++17 -lcurl -o localTime
```

---

Первый пример очень прост. Он показывает время UTC и локальное время.

#### 5.5.3.1 UTC-время и локальное время

UTC-время, или Coordinated Universal Time<sup>4</sup>, – это главный стандарт времени во всем мире. Компьютер использует время Unix<sup>5</sup>, являющееся близким приближением к UTC. Время Unix – это количество секунд, прошедших со стартовой точки Unix. Стартовое время для Unix – это 00:00:00 UTC 1 января 1970 г.

`std::chrono::system_clock::now()` возвращает в программе `localTime.cpp` время Unix.

<sup>1</sup> <https://github.com/HowardHinnant/date>.

<sup>2</sup> <https://curl.se/>.

<sup>3</sup> <https://www.iana.org/timezones>.

<sup>4</sup> [https://en.wikipedia.org/wiki/Coordinated\\_Universal\\_Time](https://en.wikipedia.org/wiki/Coordinated_Universal_Time).

<sup>5</sup> [https://en.wikipedia.org/wiki/Unix\\_time](https://en.wikipedia.org/wiki/Unix_time).

## Получение UTC-времени и локального времени

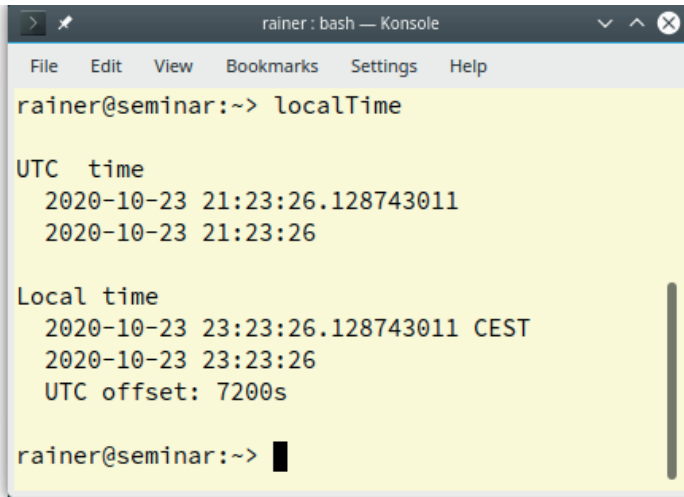
---

```

1  // localTime.cpp
2
3  #include "date/tz.h"
4  #include <iostream>
5
6  int main() {
7
8      std::cout << '\n';
9
10     using namespace date;
11
12     std::cout << "UTC time" << '\n';
13     auto utcTime = std::chrono::system_clock::now();
14     std::cout << " " << utcTime << '\n';
15     std::cout << " " << date::floor<std::chrono::seconds>(utcTime) << '\n';
16
17     std::cout << '\n';
18
19     std::cout << "Local time" << '\n';
20     auto localTime = date::make_zoned(date::current_zone(), utcTime);
21     std::cout << " " << localTime << '\n';
22     std::cout << " " << date::floor<std::chrono::seconds>(localTime.get_local_time())
23         << '\n';
24
25     auto offset = localTime.get_info().offset;
26     std::cout << " UTC offset: " << offset << '\n';
27
28     std::cout << '\n';
29
30 }
```

---

Фрагмент кода, начинающийся со строки 12, получает текущий момент времени, обрезает до секунд и показывает его. Вызов `make_zoned` (строка 20) создает `std::chrono::zoned_time localTime`. После этого вызов `localTime.get_local_time()` возвращает сохраненный момент времени как локальное время. Этот момент также усекается до секунд. `localTime` (строка 25) может быть использован для получения информации о часовом поясе. В данном случае мне интересно смещение к UTC-времени.



```

rainer@seminar:~> localTime

UTC  time
    2020-10-23 21:23:26.128743011
    2020-10-23 21:23:26

Local time
    2020-10-23 23:23:26.128743011 CEST
    2020-10-23 23:23:26
    UTC offset: 7200s

rainer@seminar:~>

```

Вывод UTC-времени и локального времени

Последний приведенный пример отвечает на важный вопрос при обучении часовым поясам: когда мне нужно начать мои занятия онлайн?

### 5.5.3.2 Различные часовые пояса для онлайн-уроков

Программа `onlineClass.cpp` отвечает на следующий вопрос: насколько это будет поздно в различных часовых поясах, если начинать уроки в 7, 13 или 17 часов локального времени (Германия)?

Онлайн-урок должен начаться 1 февраля 2021 года и продолжаться 4 часа. Из-за летнего/зимнего переноса времени нам понадобится календарная дата для получения правильного ответа.

Вычисление времени для различных часовых поясов

---

```

1  // onlineClass.cpp
2
3  #include "date/tz.h"
4  #include <algorithm>
5  #include <iomanip>
6  #include <iostream>
7
8  template <typename ZonedTime>
9  auto getMinutes(const ZonedTime& zonedTime) {
10     return date::floor<std::chrono::minutes>(zonedTime.get_local_time());
11 }
12
13 void printStartEndTimes(const date::local_days& localDay,
14                        const std::chrono::hours& h,
15                        const std::chrono::hours& durationClass,
16                        const std::initializer_list<std::string>& timeZones ){
17

```



---

```

18  date::zoned_time startDate{date::current_zone(), localDay + h};
19  date::zoned_time endDate{date::current_zone(), localDay + h + durationClass};
20  std::cout << "Local time: [" << getMinutes(startDate) << ", "
21  << getMinutes(endDate) << "]" << '\n';
22
23  longestStringSize = std::max(timeZones, [](const std::string& a,
24  < const std::string& b) { return a.size() < b.size(); }).size();
25  for (auto timeZone: timeZones) {
26  < std::cout << " " << std::setw(longestStringSize + 1) << std::left
27  << timeZone
28  << "[" << getMinutes(date::zoned_time(timeZone, startDate))
29  << ", " << getMinutes(date::zoned_time(timeZone, endDate))
30  << "]" << '\n';
31
32  }
33  }
34
35  int main() {
36
37  < using namespace std::string_literals;
38  < using namespace std::chrono;
39
40  < std::cout << '\n';
41
42  < constexpr auto classDay{date::year(2021)/2/1};
43  < constexpr auto durationClass = 4h;
44  < auto timeZones = {"America/Los_Angeles"s, "America/Denver"s,
45  < "America/New_York"s, "Europe/London"s,
46  < "Europe/Minsk"s, "Europe/Moscow"s,
47  < "Asia/Kolkata"s, "Asia/Novosibirsk"s,
48  < "Asia/Singapore"s, "Australia/Perth"s,
49  < "Australia/Sydney"s};
50
51  < for (auto startTime: {7h, 13h, 17h}) {
52  < printStartEndTimes(date::local_days{classDay}, startTime,
53  < durationClass, timeZones);
54  < std::cout << '\n';
55  }
56
57  }

```

---

Прежде чем я погружусь в детали функций `getMinutes` (строка 8) и `printStartEndTimes` (строка 13), я хотел бы сказать несколько слов о самой функции `main`. Функция `main` определяет день проведения занятия, продолжительность занятия и все часовые пояса. Цикл `for` (строка 51) выводит всю необходимую информацию.

Несколько строк (начиная со строки 18) для моих занятий `startData` и `endDate` добавляют время начала урока и его продолжительность к календарной дате. Оба значения выводятся при помощи функции `getMinutes` (строка 8). Выражение `floor<std::chrono::minutes>(zonedTime.get_local_time())` возвращает сохраненный момент времени из `std::chrono::zoned_time` и переводит его к разрешению в минутах. Для правильного выравнивания вывода программы строка 23 определяет размер самого длинного из имен часовых поясов. Строка 25 перебирает часовые пояса и выводит имя часового пояса, а также начало и конец занятия. Несколько дат даже пересекают границу дня.

```
rainer@seminar:~$ onlineClass

Local time: [2021-02-01 07:00:00, 2021-02-01 11:00:00]
America/Los_Angeles [2021-01-31 22:00:00, 2021-02-01 02:00:00]
America/Denver      [2021-01-31 23:00:00, 2021-02-01 03:00:00]
America/New_York     [2021-02-01 01:00:00, 2021-02-01 05:00:00]
Europe/London        [2021-02-01 06:00:00, 2021-02-01 10:00:00]
Europe/Minsk         [2021-02-01 09:00:00, 2021-02-01 13:00:00]
Europe/Moscow        [2021-02-01 09:00:00, 2021-02-01 13:00:00]
Asia/Kolkata         [2021-02-01 11:30:00, 2021-02-01 15:30:00]
Asia/Novosibirsk     [2021-02-01 13:00:00, 2021-02-01 17:00:00]
Asia/Singapore       [2021-02-01 14:00:00, 2021-02-01 18:00:00]
Australia/Perth      [2021-02-01 14:00:00, 2021-02-01 18:00:00]
Australia/Sydney     [2021-02-01 17:00:00, 2021-02-01 21:00:00]

Local time: [2021-02-01 13:00:00, 2021-02-01 17:00:00]
America/Los_Angeles [2021-02-01 04:00:00, 2021-02-01 08:00:00]
America/Denver      [2021-02-01 05:00:00, 2021-02-01 09:00:00]
America/New_York     [2021-02-01 07:00:00, 2021-02-01 11:00:00]
Europe/London        [2021-02-01 12:00:00, 2021-02-01 16:00:00]
Europe/Minsk         [2021-02-01 15:00:00, 2021-02-01 19:00:00]
Europe/Moscow        [2021-02-01 15:00:00, 2021-02-01 19:00:00]
Asia/Kolkata         [2021-02-01 17:30:00, 2021-02-01 21:30:00]
Asia/Novosibirsk     [2021-02-01 19:00:00, 2021-02-01 23:00:00]
Asia/Singapore       [2021-02-01 20:00:00, 2021-02-02 00:00:00]
Australia/Perth      [2021-02-01 20:00:00, 2021-02-02 00:00:00]
Australia/Sydney     [2021-02-01 23:00:00, 2021-02-02 03:00:00]

Local time: [2021-02-01 17:00:00, 2021-02-01 21:00:00]
America/Los_Angeles [2021-02-01 08:00:00, 2021-02-01 12:00:00]
America/Denver      [2021-02-01 09:00:00, 2021-02-01 13:00:00]
America/New_York     [2021-02-01 11:00:00, 2021-02-01 15:00:00]
Europe/London        [2021-02-01 16:00:00, 2021-02-01 20:00:00]
Europe/Minsk         [2021-02-01 19:00:00, 2021-02-01 23:00:00]
Europe/Moscow        [2021-02-01 19:00:00, 2021-02-01 23:00:00]
Asia/Kolkata         [2021-02-01 21:30:00, 2021-02-02 01:30:00]
Asia/Novosibirsk     [2021-02-01 23:00:00, 2021-02-02 03:00:00]
Asia/Singapore       [2021-02-02 00:00:00, 2021-02-02 04:00:00]
Australia/Perth      [2021-02-02 00:00:00, 2021-02-02 04:00:00]
Australia/Sydney     [2021-02-02 03:00:00, 2021-02-02 07:00:00]

rainer@seminar:~$
```

Вывод начала и конца занятия в различных часовых поясах

### 5.5.3.3 Новые часы

Кроме системных `std::system_clock`<sup>1</sup>, монотонных `std::steady_clock`<sup>2</sup> и наиболее точных `std::high_resolution_clock`<sup>3</sup> в C++11, в C++20 введено еще пять дополнительных часов.

- `std::utc_clock` – часы для UTC. Измеряют время, прошедшее с 00:00:00 1 января 1970 г., включая високосные секунды.
- `std::tai_clock` – часы для международного атомного времени (International Atomic Time)<sup>4</sup>. Измеряют время, прошедшее с 00:00:00 1 января 1958 г., и сдвинуты на 10 секунд по отношению к UTC-времени для этой даты. Високосные секунды не вставляются.
- `std::gps_clock` – часы для времени GPS. Они представляют часы для Global Positioning System<sup>5</sup>. Измеряют время начиная с 00:00:00 6 января 1980 г. UTC. Високосные секунды не вставляются.
- `std::file_clock` – часы для времени файла. Являются синонимом для `std::filesystem::file_time_type`<sup>6</sup>.
- `std::local_t` – псевдочасы для представления локального времени.

### 5.5.3.4 Ввод/вывод для библиотеки chrono

Благодаря функции `std::chrono::parse` и `std::formatter` из библиотеки форматирования вы можете читать и выводить объекты, представляющие время.

- `std::chrono::parse` читает объект времени из входного потока. [cppreference.com/parse](https://en.cppreference.com/parse)<sup>7</sup> содержит детальную информацию о формате входной строки.
- `std::formatter` определяет специализации для различных объектов из `chrono`. Вы можете найти детали по формату для `std::formatter` на сайте [cppreference.com/formatter](https://en.cppreference.com/formatter)<sup>8</sup>.



#### Краткая информация

- ◆ C++20 добавляет новые компоненты в библиотеку `chrono`: время дня, календарь и часовые пояса.
- ◆ Время дня – это время, прошедшее с полуночи, разделенное на часы, минуты, секунды и дробные части секунд.
- ◆ Календарь обозначает различные календарные даты, такие как год, месяц, день недели или *n*-й день недели.
- ◆ Часовой пояс представляет конкретное время в заданной географической области.

<sup>1</sup> <https://www.modernescpp.com/index.php/the-three-clocks>.

<sup>2</sup> <https://www.modernescpp.com/index.php/the-three-clocks>.

<sup>3</sup> <https://www.modernescpp.com/index.php/the-three-clocks>.

<sup>4</sup> [https://en.wikipedia.org/wiki/International\\_Atomic\\_Time](https://en.wikipedia.org/wiki/International_Atomic_Time).

<sup>5</sup> [https://en.wikipedia.org/wiki/Global\\_Positioning\\_System](https://en.wikipedia.org/wiki/Global_Positioning_System).

<sup>6</sup> [https://en.cppreference.com/w/cpp/filesystem/file\\_time\\_type](https://en.cppreference.com/w/cpp/filesystem/file_time_type).

<sup>7</sup> <https://en.cppreference.com/w/cpp/chrono/parse>.

<sup>8</sup> [https://en.cppreference.com/w/cpp/chrono/system\\_clock/formatter#Format\\_specification](https://en.cppreference.com/w/cpp/chrono/system_clock/formatter#Format_specification).

## 5.6 Библиотека форматирования



Сиппи делает чашку



### Отсутствие поддержки компиляторов

К концу 2020 года ни один из компиляторов не поддерживал библиотеку форматирования. Благодаря прототипу – библиотеке `fmt`<sup>1</sup> от Виктора Зверовича – я смог с ней поработать. Библиотека расположена на `Compiler Explorer`<sup>2</sup>. После того как один из компиляторов – GCC, Clang или MSVC – добавит поддержку библиотеки форматирования, примеры в этой главе будут обновлены.

Библиотека форматирования (Formatting Library) предлагает безопасную и расширяемую альтернативу семейству функций `printf`<sup>3</sup> и расширяет потоки ввода/вывода. Библиотека требует заголовочного файла `<format>`. Спецификации формата следуют синтаксису языка Python<sup>4</sup> и позволяют задавать заполнители и выравнивание текста, задавать знак, ширину, точность числовых значений и тип данных.

### 5.6.1 Функции форматирования

Стандарт C++20 поддерживает следующие функции форматирования.

Функции форматирования

| Функция                       | Описание                                                |
|-------------------------------|---------------------------------------------------------|
| <code>std::format</code>      | Возвращает отформатированную строку                     |
| <code>std::format_to</code>   | Передаёт результат в выходной итератор                  |
| <code>std::format_to_n</code> | Передаёт не более <i>n</i> символов в выходной итератор |

Функции форматирования принимают произвольное количество аргументов. Следующая программа `format.cpp` позволяет получить первое впечатление о функциях `std::format`, `std::format_to` и `std::format_to_n`.

<sup>1</sup> <https://github.com/fmtlib/fmt>.

<sup>2</sup> <https://godbolt.org/z/Eq5763>.

<sup>3</sup> <https://en.cppreference.com/w/cpp/io/c/fprint>.

<sup>4</sup> <https://docs.python.org/3/library/stdtypes.html#str.format>.

## Форматированный вывод

---

```
1  // format.cpp
2
3  #include <fmt/core.h>
4  #include <fmt/format.h>
5  #include <iostream>
6  #include <iterator>
7  #include <string>
8
9  int main() {
10
11     std::cout << '\n';
12
13     std::cout << fmt::format("Hello, C++{!}\n", "20") << '\n';
14
15     std::string buffer;
16
17     fmt::format_to(
18         std::back_inserter(buffer),
19         "Hello, C++{!}\n",
20         "20");
21
22     std::cout << buffer << '\n';
23
24     buffer.clear();
25
26     fmt::format_to_n(
27         std::back_inserter(buffer), 5,
28         "Hello, C++{!}\n",
29         "20");
30
31     std::cout << buffer << '\n';
32
33
34     std::cout << '\n';
35
36 }
```

---

Программа в строке 13 непосредственно выводит отформатированную строку. Вызовы в строках 17 и 26 используют строку как буфер. `std::format_to_n` помещает в буфер только пять символов.

```

1 // format.cpp
2
3 #include <format.h>
4 #include <format_args.h>
5 #include <iostream>
6 #include <string>
7
8 int main() {
9
10     std::cout << "a";
11
12     std::cout << std::format("Hello, C++{!}\n", "20");
13
14     std::string buffer;
15
16     std::format_to(
17         std::back_inserter(buffer),
18         "Hello, C++{!}\n",
19         "20");
20
21     std::cout << buffer << "a";
22
23     buffer.clear();
24 }

```

Форматированный вывод

Наиболее интересной частью использования форматирующих функций является форматная строка ("Hello, C++{!}\n").

## 5.6.2 Форматная строка

Синтаксис строки форматирования идентичен во всех функциях `std::format`, `std::format_to` и `std::format_to_n`. В моих примерах я использую функцию `std::format`.

- Синтаксис `std::format(FormatString, Args)`

Строка форматирования `FormatString` состоит из:

- обычных символов (за исключением `{}` и `}`);
- управляющих последовательностей (Escape sequences) `{{}` и `}}`, которые заменяются на `{}` и `}`;
- полей подстановки (replacement fields).

Поля для подстановки имеют формат `{}` и `}`.

- Вы можете помещать внутри поля для подстановки идентификатор аргумента и двоеточие, за которым идет описание формата, обе эти компоненты необязательны.

Идентификатор аргумента позволяет задать индекс аргумента из `Args`. Эти идентификаторы начинаются с 0. Когда вы не задаете идентификатор, то поля заполняются в том же порядке, в котором аргументы заданы в `Args`. При этом либо все заменяемые поля используют идентификаторы, либо ни одно поле не использует их, т. е. `std::format("{}, {}", "Hello", "World")` и `std::format("{1}, {0}", "World", "Hello")` скомпилируются, а вот `std::format("{1}, {}", "World", "Hello")` нет.

`std::formatter` и его спецификации определяют **спецификации формата** для типов аргументов.

- Базовые типы и `std::string`: стандартные спецификации<sup>1</sup>, основанные на спецификациях формата из Python<sup>2</sup>.
- Типы из `chrono`<sup>3</sup>.
- Другие форматируемые типы: задаваемые пользователем.

<sup>1</sup> [https://en.cppreference.com/w/cpp/utility/format/formatter#Standard\\_format\\_specification](https://en.cppreference.com/w/cpp/utility/format/formatter#Standard_format_specification).

<sup>2</sup> <https://docs.python.org/3/library/stdtypes.html#str.format>.

<sup>3</sup> [https://en.cppreference.com/w/cpp/chrono/system\\_clock/formatter#Format\\_specification](https://en.cppreference.com/w/cpp/chrono/system_clock/formatter#Format_specification).

В следующих разделах я буду на практике показывать применение всего этого. Давайте начнем с идентификаторов аргументов и потом перейдем к спецификаторам формата.

### 5.6.2.1 Идентификатор аргумента

Благодаря идентификаторам аргументов вы можете переупорядочить аргументы или обратиться к конкретным аргументам.

Использование идентификатора аргумента

---

```

1 // formatArgumentID.cpp
2
3 #include <fmt/core.h>
4 #include <iostream>
5 #include <string>
6
7 int main() {
8
9     std::cout << '\n';
10
11     std::cout << fmt::format("{} {}: {}!\n", "Hello", "World", 2020);
12
13     std::cout << fmt::format("{1} {0}: {2}!\n", "World", "Hello", 2020);
14
15     std::cout << fmt::format("{0} {0} {1}: {2}!\n", "Hello", "World", 2020);
16
17     std::cout << fmt::format("{0}: {2}!\n", "Hello", "World", 2020);
18
19     std::cout << '\n';
20
21 }
```

---

Строка 11 выводит аргументы в том порядке, в котором они были заданы. Строка 13 меняет местами первый и второй аргументы, строка 15 дважды показывает первый аргумент, и строка 17 игнорирует второй аргумент.

Вывод программы:

```

Hello World: 2020!
Hello World: 2020!
Hello Hello World: 2020!
Hello: 2020!
```

Применение идентификаторов аргументов

Применение идентификаторов вместе со спецификаторами формата делает форматирование текста в стандарте C++20 очень мощным.

### 5.6.2.2 Спецификация формата

Я не буду приводить спецификации формата для базовых типов, строк или типов из `chrono`. О базовых типах и строках вы можете получить всю информацию в стандартных спецификациях<sup>1</sup>. Аналогично вы можете получить всю информацию о спецификациях для типов из `chrono` в соответствующих спецификациях<sup>2</sup>.

Вместо этого я приведу упрощенную спецификацию формата для базовых типов и строк.

Упрощенная спецификация формата для базовых типов и строк имеет следующий вид:

Все ее части не обязательны. Следующие несколько разделов объяснят различные части из этой спецификации формата.

#### 5.6.2.2.1 Символ заполнения и выравнивание

Символ заполнения (fill character) не обязателен (любой символ, кроме { и }), и за ним следует спецификатор выравнивания.

- Символ заполнения: по умолчанию используется пробел.
- Выравнивание:
  - ◆ < слева (по умолчанию для нечисловых значений);
  - ◆ > справа (по умолчанию для чисел);
  - ◆ ^ по центру.

Применение символа заполнения и выравнивания

---

```
// formatFillAlign.cpp
```

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

```
#include <iostream>
```

```
int main() {
```

```
    std::cout << '\n';
```

```
    int num = 2020;
```

```
    std::cout << fmt::format("{:6}", num) << '\n';
```

```
    std::cout << fmt::format("{:6}", 'x') << '\n';
```

```
    std::cout << fmt::format("{:*<6}", 'x') << '\n';
```

---

<sup>1</sup> [https://en.cppreference.com/w/cpp/utility/format/formatter#Standard\\_format\\_specification](https://en.cppreference.com/w/cpp/utility/format/formatter#Standard_format_specification).

<sup>2</sup> [https://en.cppreference.com/w/cpp/chrono/system\\_clock/formatter#Format\\_specification](https://en.cppreference.com/w/cpp/chrono/system_clock/formatter#Format_specification).

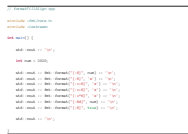


```

std::cout << fmt::format("{:*>6}", 'x') << '\n';
std::cout << fmt::format("{:*^6}", 'x') << '\n';
std::cout << fmt::format("{:6d}", num) << '\n';
std::cout << fmt::format("{:6}", true) << '\n';

std::cout << '\n';
}

```



```

x
x
123456
true

```

Применение символа заполнения и выравнивания

#### 5.6.2.2.2 Знак, # и 0

Знак, # и 0 допустимы только тогда, когда используются целый и вещественный типы данных.

Знак может принимать следующие значения:

- +: знак используется для нуля и положительных чисел;
- -: знак используется только для отрицательных чисел (по умолчанию);
- пробел: пробел вначале используется для неотрицательных чисел, и минус используется для отрицательных.

Применение знака

```

// formatSign.cpp

#include <fmt/core.h>
#include <iostream>

int main() {

    std::cout << '\n';

```

```
std::cout << std::format("{0:},{0:+},{0:-},{0: }", 0) << '\n';
std::cout << std::format("{0:},{0:+},{0:-},{0: }", -0) << '\n';
std::cout << std::format("{0:},{0:+},{0:-},{0: }", 1) << '\n';
std::cout << std::format("{0:},{0:+},{0:-},{0: }", -1) << '\n';

std::cout << '\n';

}
```

---

```
// formatSign.cpp
#include <fmt/core.h>
#include <iostream>

int main() {
    fmt::cout << "0";
    fmt::cout << fmt::format("{:010}", 0) << '\n';
    fmt::cout << fmt::format("{:010}", -0) << '\n';
    fmt::cout << fmt::format("{:010}", 1) << '\n';
    fmt::cout << fmt::format("{:010}", -1) << '\n';
    fmt::cout << '\n';
}
```

### Применение знака

Символ # вызывает альтернативный вид:

- для целых типов для бинарных, восьмеричных и шестнадцатеричных типов используется 0b, 0 и 0x;
- для чисел с плавающей точкой всегда используется десятичная точка;
- 0: заполнитель в виде нулей.

```
1 // formatAlternate.cpp
2
3 #include <fmt/core.h>
4 #include <iostream>
5
6 int main() {
7
8     std::cout << '\n';
9
10    std::cout << fmt::format("{:#015}", 0x78) << '\n';
11    std::cout << fmt::format("{:#015b}", 0x78) << '\n';
12    std::cout << fmt::format("{:#015x}", 0x78) << '\n';
13
14    std::cout << '\n';
15
16    std::cout << fmt::format("{:g}", 120.0) << '\n';
```

```

17     std::cout << fmt::format("{:g}", 120.0) << '\n';
18
19
20     std::cout << '\n';
21
22 }

```

```

1 // formatAlternate.cpp
2
3 #include <fmt/core.h>
4 #include <iostream>
5
6 int main() {
7
8     std::cout << '\n';
9
10    std::cout << fmt::format("{:015}", 0x78) << '\n';
11    std::cout << fmt::format("{:015b}", 0x78) << '\n';
12    std::cout << fmt::format("{:015x}", 0x78) << '\n';
13
14    std::cout << '\n';
15
16    std::cout << fmt::format("{:g}", 120.0) << '\n';
17    std::cout << fmt::format("{:g}", 120.0) << '\n';
18
19 }

```

### Применение # и 0

#### 5.6.2.2.3 Ширина и точность

Вы можете задать ширину и точность для вашего типа. Спецификатор ширины может применяться к числам, а точность – к числам с плавающей точкой и строкам. Для типов с плавающей точкой точность задает точность форматирования; для строк точность позволяет задать, сколько символов используется, фактически обрезая строку. Если точность больше длины строки, то никакого влияния она не оказывает.

- **Ширина:** вы можете задать положительное десятичное число либо заменяемое поле ({ } или {n}). Если задано n, то оно определяет минимальную ширину поля.
- **Точность:** вы можете использовать точку (.), за которой идет либо неотрицательное десятичное число, либо заменяемое поле.

Ниже приведено несколько примеров, как это использовать на практике.

Применение спецификаторов ширины и точности

```

1 // formatWidthPrecision.cpp
2
3 #include <fmt/core.h>
4 #include <iostream>
5 #include <string>
6
7 int main() {
8
9     int i = 123456789;
10    double d = 123.456789;
11

```

```
12 std::cout << "----" << fmt::format("{:}", i) << "----\n";
13 std::cout << "----" << fmt::format("{:15}", i) << "----\n"; // (w = 15)
14 std::cout << "----" << fmt::format("{:}", i, 15) << "----\n"; // (w = 15)
15
16 std::cout << '\n';
17
18 std::cout << "----" << fmt::format("{:}", d) << "----\n";
19 std::cout << "----" << fmt::format("{:15}", d) << "----\n"; // (w = 15)
20 std::cout << "----" << fmt::format("{:}", d, 15) << "----\n"; // (w = 15)
21
22 std::cout << '\n';
23
24 std::string s = "Only a test";
25
26 std::cout << "----" << fmt::format("{:10.50}", d) << "----\n"; // (w = 50, p = 50)
27 std::cout << "----" << fmt::format("{:{}.{} }", d, 10, 50) << "----\n"; // (w = 50,
28 // p = 50)
29 std::cout << "----" << fmt::format("{:10.5}", d) << "----\n"; // (w = 10, p = 5)
30 std::cout << "----" << fmt::format("{:{}.{} }", d, 10, 5) << "----\n"; // (w = 10,
31 // p = 5)
32
33 std::cout << '\n';
34
35 std::cout << "----" << fmt::format("{:.500}", s) << "----\n"; // (p = 500)
36 std::cout << "----" << fmt::format("{:{}.{} }", s, 500) << "----\n"; // (p = 500)
37 std::cout << "----" << fmt::format("{:.5}", s) << "----\n"; // (p = 5)
38
39 }
```

---

Символ *w* в исходном коде обозначает ширину, символ *p* – точность. Есть несколько интересных замечаний по этой программе. Когда вы задаете ширину при помощи заменяемого поля (строка 14), то не добавляется дополнительных пробелов. Когда вы задаете точность выше, чем длина `double` (строки 26 и 27), длина выводимого значения отражает точность. Это не работает для строк (строки 35 и 36).

```

1 // formatWidthPrecision.cpp
2
3 #include <fmt/core.h>
4 #include <iostream>
5 #include <string>
6
7 int main() {
8
9     int i = 123456789;
10    double d = 123.456789;
11
12    std::cout << "----" << fmt::format("{ }", i) << "----\n";
13    std::cout << "----" << fmt::format("{:15}", i) << "----\n"; // (w = 15)
14    std::cout << "----" << fmt::format("{:}", i, 15) << "----\n"; // (w = 15)
15
16    std::cout << '\n';
17
18    std::cout << "----" << fmt::format("{ }", d) << "----\n";
19    std::cout << "----" << fmt::format("{:15}", d) << "----\n"; // (w = 15)
20    std::cout << "----" << fmt::format("{:}", d, 15) << "----\n"; // (w = 15)
21
22    std::cout << '\n';

```

Применение спецификаторов ширины и точности

#### 5.6.2.2.4 Тип

В общем случае компилятор сам выводит тип используемого значения. Но иногда вам нужно явно его указать. Ниже приводятся наиболее важные спецификаторы типа.

- Строки: s.
- Целые числа:
  - ◆ b: двоичный вид;
  - ◆ B: то же, что и b, но с префиксом 0B;
  - ◆ d: десятичный формат;
  - ◆ o: восьмеричный формат;
  - ◆ x: шестнадцатеричный формат;
  - ◆ X: то же, что и x, но с префиксом 0X.
- char и wchar\_t:
  - ◆ b, B, d, o, x, X как для целых чисел.
- bool:
  - ◆ s: true или false;
  - ◆ b, B, d, o, x, X как целые числа.
- вещественный тип:
  - ◆ e: экспоненциальный формат;
  - ◆ E: как и e, но экспонента пишется через E;
  - ◆ f, F: фиксированная точка, точность равна 6;
  - ◆ g, G: точность 6, но экспонента пишется через E.

Когда вы не задаете тип, то значения выводятся следующим образом: строка выводится как строка, целое число выводится в десятичном виде, символ как символ, а значение с плавающей точкой выводится при помощи `std::to_chars`<sup>1</sup>.

Благодаря спецификаторам типа вы можете выводить `int` в различных системах счисления.

Применение спецификатора типа

---

```

1  // formatType.cpp
2
3  #include <fmt/core.h>
4  #include <iostream>
5
6  int main() {
7
8      int num{2020};
9
10     std::cout << "default:      " << fmt::format("{:}", num) << '\n';
11     std::cout << "decimal:      " << fmt::format("{:d}", num) << '\n';
12     std::cout << "binary:       " << fmt::format("{:b}", num) << '\n';
13     std::cout << "octal:        " << fmt::format("{:o}", num) << '\n';
14     std::cout << "hexadecimal: " << fmt::format("{:x}", num) << '\n';
15
16 }
```

---



---

```

1  // formatType.cpp
2
3  #include <fmt/core.h>
4  #include <iostream>
5
6  int main() {
7
8      int num{2020};
9
10     std::cout << "default:      " << fmt::format("{:}", num) << '\n';
11     std::cout << "decimal:      " << fmt::format("{:d}", num) << '\n';
12     std::cout << "binary:       " << fmt::format("{:b}", num) << '\n';
13     std::cout << "octal:        " << fmt::format("{:o}", num) << '\n';
14     std::cout << "hexadecimal: " << fmt::format("{:x}", num) << '\n';
15
16 }
```

---

Применение спецификатора типа

До сих пор я форматировал базовые типы и строки. Но вы можете также форматировать и задаваемые пользователем типы.

### 5.6.3 Задаваемые пользователем типы

Для того чтобы отформатировать задаваемый пользователем тип, необходимо специализировать класс `std::formatter`<sup>2</sup> для этого класса. Это значит, что нужно реализовать функции `parse` и `format`.

<sup>1</sup> [https://en.cppreference.com/w/cpp/utility/to\\_chars](https://en.cppreference.com/w/cpp/utility/to_chars).

<sup>2</sup> <https://en.cppreference.com/w/cpp/utility/format/formatter>.

- `parse`:
  - ◆ принимает на вход контекст парсинга;
  - ◆ разбирает этот контекст;
  - ◆ возвращает итератор на конец спецификации формата;
  - ◆ генерирует исключение `std::format_error` в случае ошибки.
- `format`:
  - ◆ получает на вход значение `t`, которое должно быть отформатировано, и контекст форматирования `fc`;
  - ◆ форматирует `t` в соответствии с `fc`;
  - ◆ записывает результат в `fc.out()`;
  - ◆ возвращает итератор, представляющий конец вывода.

Давайте теперь перейдем от теории к практике и отформатируем `std::vector`.

### 5.6.3.1 Форматируем `std::vector`

Моя первая специализация класса `std::formatter` очень проста. Я задаю спецификацию формата для каждого элемента контейнера.

Применение спецификации формата к элементам `std::vector`

---

```

1 // formatVector.cpp
2
3 #include <iostream>
4 #include <fmt/format.h>
5 #include <string>
6 #include <vector>
7
8 template <typename T>
9 struct fmt::formatter<std::vector<T>> {
10
11     std::string formatString;
12
13     auto constexpr parse(format_parse_context& ctx) {
14         formatString = "{:";
15         std::string parseContext(std::begin(ctx), std::end(ctx));
16         formatString += parseContext;
17         return std::end(ctx) - 1;
18     }
19
20     template <typename FormatContext>
21     auto format(const std::vector<T>& v, FormatContext& ctx) {
22         auto out= ctx.out();
23         fmt::format_to(out, "[";
24         if (v.size() > 0) fmt::format_to(out, formatString, v[0]);
25         for (int i=1; i < v.size(); ++i) fmt::format_to(out, ", " + formatString, v[i]);
26         fmt::format_to(out, "]");
27         return fmt::format_to(out, "\n" );
28     }
29

```

```
30 };
31
32
33 int main() {
34
35     std::vector<int> myInts{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
36     std::cout << fmt::format("{:}", myInts);
37     std::cout << fmt::format("{:+}", myInts);
38     std::cout << fmt::format("{:03d}", myInts);
39     std::cout << fmt::format("{:b}", myInts);
40
41     std::cout << '\n';
42
43     std::vector<std::string> myStrings{"Only", "for", "testing", "purpose"};
44     std::cout << fmt::format("{:}", myStrings);
45     std::cout << fmt::format("{:.3}", myStrings);
46
47 }
```

---

Специализация для `std::vector` (строка 8) содержит метод `parse` (строка 13) и метод `format` (строка 20). Метод `parse` создает `formatString`, которая применяется к каждому элементу `std::vector` (строки 25 и 26). Контекст `ctx` (строка 13) содержит символы между двоеточием (:) и закрывающей фигурной скобкой (}). В конце функция возвращает итератор на закрывающую скобку (}). Задача функции `format` более интересна. Контекст возвращает итератор вывода. Благодаря этому итератору и функции `std::format_to`<sup>1</sup> элементы `std::vector` выводятся в красивом виде.

Элементы `std::vector` (строка 35) форматируются несколькими способами. Строка 36 выводит значение, строка 37 выводит знак перед числом, строка 38 выравнивает их до 3 символов и использует 0 как заполнитель. Строка 39 выводит их в двоичном формате. Оставшиеся две строки выводят каждую строку `std::vector`. Строка 45 обрезает каждую строку до 3 символов.

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[+1, +2, +3, +4, +5, +6, +7, +8, +9, +10]
[001, 002, 003, 004, 005, 006, 007, 008, 009, 010]
[1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010]

[Only, for, testing, purpose]
[Onl, for, tes, pur]
```

Применение спецификации формата к элементам `std::vector`

Когда `std::vector` становится больше, то я хочу добавить разрыв строки (`linebreak`). Для этого случая я расширяю синтаксис спецификации формата.

---

<sup>1</sup> [https://en.cppreference.com/w/cpp/utility/format/format\\_to](https://en.cppreference.com/w/cpp/utility/format/format_to).



Размещение элементов `std::vector`


---

```

1  // formatVectorLinebreak.cpp
2
3  #include <algorithm>
4  #include <iostream>
5  #include <limits>
6  #include <numeric>
7  #include <fmt/format.h>
8  #include <string>
9  #include <vector>
10
11 template <typename T>
12 struct fmt::formatter<std::vector<T>> {
13
14     std::string systemFormatString;
15     std::string userFormatString;
16     int lineBreak{std::numeric_limits<int>::max()};
17
18     auto constexpr parse(format_parse_context& ctx) {
19         std::string startFormatString = "{:";
20         std::string parseContext(std::begin(ctx), std::end(ctx));
21         auto posCurly = parseContext.find_last_of("{}");
22         auto posTab = parseContext.find_last_of("|");
23         if (posTab == std::string::npos) {
24             systemFormatString = startFormatString + parseContext.substr(0, posCurly + 1);
25         }
26         else {
27             systemFormatString = startFormatString + parseContext.substr(0, posTab) + ":";
28             userFormatString = parseContext.substr(posTab + 1, posCurly - posTab - 1);
29             lineBreak = std::stoi(userFormatString);
30         }
31         return std::begin(ctx) + posCurly;
32     }
33
34     template <typename FormatContext>
35     auto format(const std::vector<T>& v, FormatContext& ctx) {
36         auto out = ctx.out();
37         auto vectorSize = v.size();
38         if (vectorSize == 0) return fmt::format_to(out, "\n");
39         for (int i = 1; i < vectorSize + 1; ++i) {
40             fmt::format_to(out, systemFormatString, v[i-1]);
41             if ( ( i % lineBreak ) == 0 ) fmt::format_to(out, "\n");
42         }
43         return fmt::format_to(out, "\n" );
44     }
45 };
46
47

```

```
48 int main() {  
49  
50     std::vector<int> myInts(100);  
51     std::iota(myInts.begin(), myInts.end(), 1);  
52  
53     std::cout << fmt::format("{:|20}", myInts);  
54     std::cout << '\n';  
55     std::cout << fmt::format("{: |20}", myInts);  
56     std::cout << '\n';  
57     std::cout << fmt::format("{:4d|20}", myInts);  
58     std::cout << '\n';  
59     std::cout << fmt::format("{:10b|8}", myInts);  
60  
61 }
```

---

Разберемся, как это работает. Я поддерживаю в спецификации формата необязательный символ `|`, за которым следует число. Число говорит о том, нужно ли добавить разрыв строки. Я ищу символ `|` и закрывающую фигурную скобку `}`. По соображениям быстродействия я начинаю в строках 21 и 22 с конца. Благодаря индексу символа `|` и индексу `}` я могу создать строки `systemFormatString` и `useFormatString` (строки с 24 по 29). Метод `format` использует `systemFormatString` и применяет ее к каждому элементу вектора. Я добавляю разрыв строки, когда выполнено (`i % lineBreak==0`) (строка 41).

Строка 53 выводит 20 элементов в одной строке и вставляет разрыв строки. Можно сделать лучше. Спецификация формата `{: |20}` (строка 55) добавляет пробел перед каждым числом. Кроме того, строка 57 выравнивает каждый элемент по 4 символам. Последняя строка выводит по 8 элементов на строку, выравнивая каждый элемент по 8 символам `{:10b|8}`.

Следующий скриншот показывает читаемые отформатированные элементы `std::vector`.

```
1234567891011121314151617181920
2122232425262728293031323334353637383940
4142434445464748495051525354555657585960
6162636465666768697071727374757677787980
81828384858687888990919293949596979899100
```

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
```

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
```

```
1      10      11      100      101      110      111      1000
1001    1010    1011    1100    1101    1110    1111    10000
10001    10010    10011    10100    10101    10110    10111    11000
11001    11010    11011    11100    11101    11110    11111    100000
100001    100010    100011    100100    100101    100110    100111    101000
101001    101010    101011    101100    101101    101110    101111    110000
110001    110010    110011    110100    110101    110110    110111    111000
111001    111010    111011    111100    111101    111110    111111    1000000
1000001    1000010    1000011    1000100    1000101    1000110    1000111    1001000
1001001    1001010    1001011    1001100    1001101    1001110    1001111    1010000
1010001    1010010    1010011    1010100    1010101    1010110    1010111    1011000
1011001    1011010    1011011    1011100    1011101    1011110    1011111    1100000
1100001    1100010    1100011    1100100
```

Применение спецификаций формата и разрывов строк к элементам `std::vector`



### Краткая информация

- ♦ Библиотека форматирования предоставляет безопасную и расширяемую альтернативу семейству `printf` и расширяет потоки ввода/вывода.
- ♦ Спецификация формата позволяет задавать символы заполнения и выравнивание текста, а также знак, ширину и точность чисел и указывать тип данных.
- ♦ Благодаря функциям `parse` и `format` вы можете настроить под себя форматирование задаваемых пользователем типов данных.

## 5.7 Дальнейшие улучшения



Сиппи поднимается вверх

### 5.7.1 `std::bind_front`

Функция `std::bind_front` (`Func&& func, Args&& ... args`) создает вызываемую обертку (wrapper) для функции `func`. У `std::bind_front` может быть произвольное количество аргументов.



#### Различие `std::bind` и `std::bind_front`

Начиная со стандарта C++11 у нас есть `std::bind`<sup>1</sup> и лямбды<sup>2</sup>. Начиная со стандарта C++20 мы получили `std::bind_front`<sup>3</sup>. Это может вызвать некоторое удивление. Строго говоря, `std::bind` доступна начиная даже с Technical Report 1<sup>4</sup> (TR1). `std::bind` и лямбды могут выступать как замена `std::bind_front`. Более того, `std::bind_front` кажется младшей сестрой `std::bind`, поскольку только `std::bind` поддерживает переупорядочивание аргументов. Конечно же, есть причина, чтобы в дальнейшем использовать `std::bind_front`: в отличие от `std::bind`, функция `std::bind_front` распространяет (propagate) исключение до вызывающего оператора.

Следующая программа показывает вам, что вы можете заменить `std::bind_front` при помощи `std::bind` или лямбды.

<sup>1</sup> <https://en.cppreference.com/w/cpp/utility/functional/bind>.

<sup>2</sup> <https://en.cppreference.com/w/cpp/language/lambda>.

<sup>3</sup> [https://en.cppreference.com/w/cpp/utility/functional/bind\\_front](https://en.cppreference.com/w/cpp/utility/functional/bind_front).

<sup>4</sup> [https://en.wikipedia.org/wiki/C%2B%2B\\_Technical\\_Report\\_1](https://en.wikipedia.org/wiki/C%2B%2B_Technical_Report_1).

Сравнение `std::bind_front`, `std::bind` и лямбды

---

```

1  // bindFront.cpp
2
3  #include <functional>
4  #include <iostream>
5
6  int plusFunction(int a, int b) {
7      return a + b;
8  }
9
10 auto plusLambda = [](int a, int b) {
11     return a + b;
12 };
13
14 int main() {
15
16     std::cout << '\n';
17
18     auto twoThousandPlus1 = std::bind_front(plusFunction, 2000);
19     std::cout << "twoThousandPlus1(20): " << twoThousandPlus1(20) << '\n';
20
21     auto twoThousandPlus2 = std::bind_front(plusLambda, 2000);
22     std::cout << "twoThousandPlus2(20): " << twoThousandPlus2(20) << '\n';
23
24     auto twoThousandPlus3 = std::bind_front(std::plus<int>(), 2000);
25     std::cout << "twoThousandPlus3(20): " << twoThousandPlus3(20) << '\n';
26
27     std::cout << "\n\n";
28
29     using namespace std::placeholders;
30
31     auto twoThousandPlus4 = std::bind(plusFunction, 2000, _1);
32     std::cout << "twoThousandPlus4(20): " << twoThousandPlus4(20) << '\n';
33
34     auto twoThousandPlus5 = [](int b) { return plusLambda(2000, b); };
35     std::cout << "twoThousandPlus5(20): " << twoThousandPlus5(20) << '\n';
36
37     std::cout << '\n';
38
39 }
```

---

Каждый вызов (строки 18, 21, 24, 31 и 34) получает вызываемый объект, принимающий два аргумента, и возвращает вызываемый объект, принимающий только один аргумент, поскольку первый аргумент задан равным 2000. Вызываемый объект – это функция (строка 21) и предопределенный функциональный объект (строка 24). Параметр `_1` – это так называемый заполнитель (placeholder) (строка 31), обозначающий пропущенный аргумент. При помощи лямбд (строка 34) вы можете явно применить один аргумент и передать аргумент `b` в качестве пропущенного. С точки зрения читаемости `std::bind_front` может быть легче для чтения, чем `std::bind` или лямбда.

```
twoThousandPlus1(20) : 2020
twoThousandPlus2(20) : 2020
twoThousandPlus3(20) : 2020

twoThousandPlus4(20) : 2020
twoThousandPlus5(20) : 2020
```

Применение `std::bind_front`, `std::bind` и лямбды

## 5.7.2 `std::is_constant_evaluated`

Функция `std::is_constant_evaluated` определяет, реализуется функция во время компиляции или выполнения. Зачем нам нужна эта функция из библиотеки свойств типов (type traits)? В стандарте C++20 существует три типа функций:

- `constexpr` – объявленные как выполняемые во время компиляции: `constexpr int alwaysCompiletime();`
- `constexpr` – могут выполняться как во время компиляции, так и во время выполнения: `constexpr int itDepends();`
- обычные функции, исполняемые во время выполнения программы: `int alwaysRuntime().`

Теперь мне нужно написать о сложном случае: `constexpr`. Такая функция может выполняться во время выполнения программы или во время компиляции. Иногда эти функции должны вести себя по-разному в зависимости от того, выполняются ли они во время компиляции или во время выполнения. У `constexpr`-функции `getSum` есть возможность выполняться во время компиляции.

Функция, объявленная как `constexpr`

---

```
constexpr int getSum(int l, int r) {
    return l + r;
}
```

---

Как мы можем быть уверены, что эта функция выполняется во время компиляции? На самом деле есть три возможности.

1. Функция, объявленная как `constexpr`, выполняется во время компиляции:
  - ♦ функция используется в контексте константного выполнения (`constant-evaluated context`). Такой контекст может быть внутри `constexpr`-функции или `static_assert`;
  - ♦ пользователь функции явно хочет получить результат во время компиляции: `constexpr auto res = getSum(2000, 11)`. Теперь `getSum()` должна выполняться во время компиляции;
2. `constexpr`-функция может быть выполнена во время выполнения программы, только если ее аргументы не `constexpr`. Это может быть случай `getSum(a, 11)`, когда переменная `a` не объявлена как `constexpr`: `int a = 2000`;
3. `constexpr`-функция может быть выполнена во время компиляции или выполнения, если не применимо ни правило 1, ни правило 2. В этом случае корректны оба варианта и компилятор сам принимает решение.

Именно пункт 3 – это то самое место, где работает функция `std::is_constant_evaluated`. Вы можете определить, выполняется ли она во время компиляции или выполнения, и вести себя по-разному. [cppreference.com/is\\_constant\\_evaluated](https://en.cppreference.com/is_constant_evaluated)<sup>1</sup> показывает как раз такой случай. Во время компиляции вы можете вычислить степень числа явно; во время выполнения вы вызываете функцию `pow`.

Выполнение различного кода во время компиляции и во время выполнения программы

---

```
// constantEvaluated.cpp
```

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

constexpr double power(double b, int x) {
    if (std::is_constant_evaluated() && !(b == 0.0 && x < 0)) {

        if (x == 0)
            return 1.0;
        double r = 1.0, p = x > 0 ? b : 1.0 / b;
        auto u = unsigned(x > 0 ? x : -x);
        while (u != 0) {
            if (u & 1) r *= p;
            u /= 2;
            p *= p;
        }
        return r;
    }
}
```

---

<sup>1</sup> [https://en.cppreference.com/w/cpp/types/is\\_constant\\_evaluated](https://en.cppreference.com/w/cpp/types/is_constant_evaluated).

```
    else {  
        return std::pow(b, double(x));  
    }  
}  
  
int main() {  
  
    std::cout << '\n';  
  
    constexpr double kilo1 = power(10.0, 3);  
    std::cout << "kilo1: " << kilo1 << '\n';  
  
    int n = 3;  
    double kilo2 = power(10.0, n);  
    std::cout << "kilo2: " << kilo2 << '\n';  
  
    std::cout << '\n';  
  
}
```

---

Следует отметить, что `std::is_constant_evaluated` можно использовать в `constexpr`-функции или внутри функции, которая может выполняться только во время выполнения программы. Конечно, их результат будет всегда `true` или `false`.

### 5.7.3 `std::source_location`

Класс `std::source_location` возвращает информацию об исходном коде. Эта информация включает в себя имена файлов, номера строк и имена функций. Эта информация очень важна при отладке, логгировании и тестировании разрабатываемых программ. Класс `std::source_location` является более удачной альтернативой макросам `__FILE__` и `__LINE__`, и лучше вместо них использовать его.

Класс `std::source_location` может дать вам следующую информацию:

| Функция                                      | Описание                                                                   |
|----------------------------------------------|----------------------------------------------------------------------------|
| <code>std::source_location::current()</code> | Создает новый экземпляр <code>std::source_location</code> <code>src</code> |
| <code>src.line()</code>                      | Возвращает номер строки                                                    |
| <code>src.column()</code>                    | Возвращает позицию                                                         |
| <code>src.file_name()</code>                 | Возвращает имя файла                                                       |
| <code>src.function_name()</code>             | Возвращает имя функции                                                     |



Вызов `std::source_location::current()` создает новый объект `src`, представляющий информацию о месте вызова. На конец 2020 года ни один компилятор не поддерживал `std::source_location`. Поэтому следующая программа взята из [cppreference.com/source\\_location](https://en.cppreference.com/source_location)<sup>1</sup>.

Вывод информации о месте вызова при помощи `std::source_location`

---

```

1 // sourceLocation.cpp
2 // from cppreference.com
3
4 #include <iostream>
5 #include <string_view>
6 #include <source_location>
7
8 void log(std::string_view message,
9         const std::source_location& location = std::source_location::current())
10 {
11     std::cout << "info:"
12               << location.file_name() << ':'
13               << location.line() << ' '
14               << message << '\n';
15 }
16
17 int main()
18 {
19     log("Hello world!"); // info:main.cpp:19 Hello world!
20 }
```

---



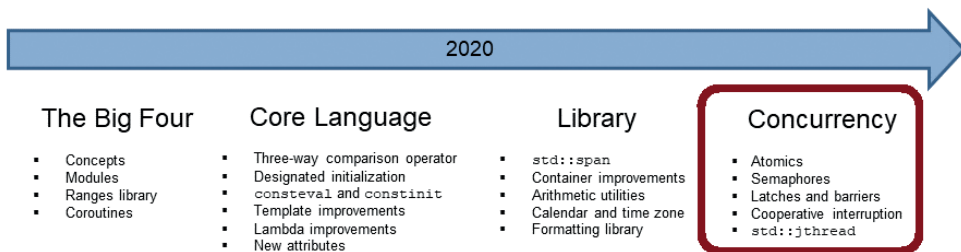
### Краткая информация

- ♦ `std::bind_front` – это более удобная альтернатива `std::bind` (появившейся в стандарте C++11). В отличие от `std::bind`, функция `std::bind_front` не позволяет переупорядочение своих аргументов.
- ♦ Функция `std::is_constant_evaluated` определяет, вызывается функция во время компиляции или во время выполнения.
- ♦ `std::source_location` содержит информацию об исходном коде. Эта информация включает в себя имена файлов, номера строк и имена функций и может использоваться для отладки, логгирования и тестирования программ.

<sup>1</sup> [https://en.cppreference.com/w/cpp/utility/source\\_location](https://en.cppreference.com/w/cpp/utility/source_location).

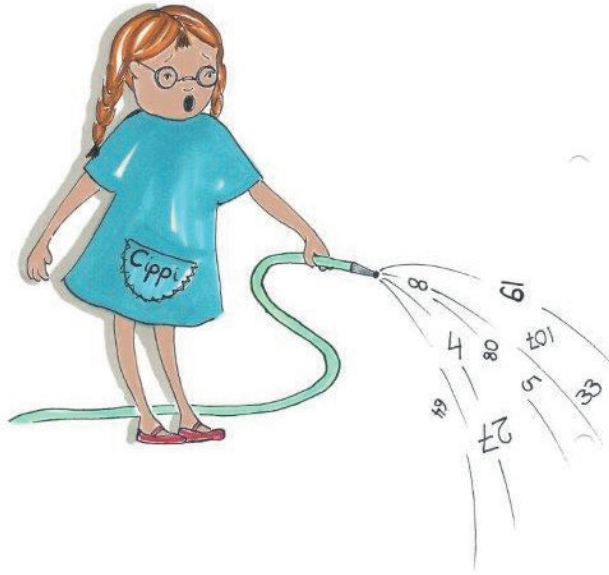
## 6. Параллельность

### C++20



С публикацией стандарта C++11 в C++ появились многопоточная библиотека и многопоточная модель памяти. Эта библиотека содержит базовые строительные блоки, такие как атомарные переменные, потоки и др. Это та основа, на которой стандарты C++, такие как C++20, могут строить более высокоуровневые абстракции.

## 6.1 Корутины



Сиппи поливает цветы

Корутины (coroutines, сопрограммы) – это функции, которые могут приостанавливать и продолжать свое выполнение, сохраняя при этом свое состояние. Эволюция функций в C++ сделала шаг вперед.



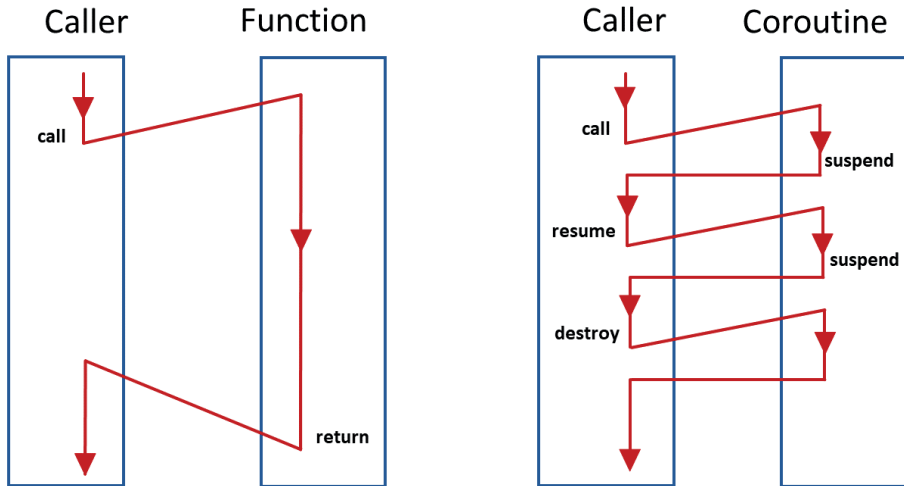
### Сложность понимания сопрограмм

Для меня было сложно понять сопрограммы. Я настоятельно рекомендую вам не читать эти разделы последовательно. Пропустите следующие два раздела. Прочтите примеры к разделам начиная с «Вариации объектов future» и т.д. Чтение, изучение и работа с предоставленными примерами должны дать вам начальное желание углубиться в детали и то, как устроены сопрограммы.

То, что я представляю в этой главе как новую идею в C++, на самом деле довольно старо. Сам термин «корутин» (сoproграмма) был придуман Мелвином Конвеем<sup>1</sup>. Он использовал его в своей работе по созданию компилятора в 1963 г. Дональд Кнут<sup>2</sup> назвал процедуры частным случаем сопрограмм. Просто иногда нужно время, чтобы ваши идеи приняли.

<sup>1</sup> [https://en.wikipedia.org/wiki/Melvin\\_Conway](https://en.wikipedia.org/wiki/Melvin_Conway).

<sup>2</sup> [https://en.wikipedia.org/wiki/Donald\\_Knuth](https://en.wikipedia.org/wiki/Donald_Knuth).



Функции по сравнению с сопрограммами

В то время как вы можете только вызвать функцию и вернуть из нее значение, с сопрограммой все иначе – вы можете вызвать ее, приостановить ее выполнение (`suspend`) и продолжить его (`resume`), а также уничтожить приостановленную сопрограмму.

С новыми ключевыми словами `co_await` и `co_yield` стандарт C++20 расширяет выполнение функций в C++ двумя новыми понятиями.

Благодаря `co_await expression` стало возможным приостановить и продолжить выполнение `expression`. Если вы используете `co_await expression` в функции `func`, то вызов `auto getResult = func()` не блокируется, если результат функции не готов. Вместо дорогостоящей блокировки и у вас есть более экономное ожидание.

Конструкция `co_yield expression` поддерживает функции-генераторы. Такая функция возвращает новое значение каждый раз, когда вы ее вызываете. Функция-генератор – это что-то вроде потока данных, из которого вы можете брать значения. Сам поток может быть бесконечным. Тем самым мы подошли к самому сердцу отложенного выполнения (*lazy evaluation*) в C++.

### 6.1.1 Функция-генератор

Следующая программа максимально проста. Функция `getNumbers` возвращает все целые числа от `begin` до `end` с шагом `inc`. Значение `begin` должно быть меньше, чем `end`, и `inc` должно быть положительным.

Жадная (*greedy*) функция-генератор

```
1 // greedyGenerator.cpp
2
3 #include <iostream>
4 #include <vector>
5
```

```

6  std::vector<int> getNumbers(int begin, int end, int inc = 1) {
7
8      std::vector<int> numbers;
9      for (int i = begin; i < end; i += inc) {
10         numbers.push_back(i);
11     }
12
13     return numbers;
14
15 }
16
17 int main() {
18
19     std::cout << '\n';
20
21     const auto numbers= getNumbers(-10, 11);
22
23     for (auto n: numbers) std::cout << n << " ";
24
25     std::cout << "\n\n";
26
27     for (auto n: getNumbers(0, 101, 5)) std::cout << n << " ";
28
29     std::cout << "\n\n";
30
31 }

```

Конечно, здесь я изобретаю колесо, поскольку эта работа могла быть выполнена при помощи `std::iota`<sup>1</sup>.

Ниже приводится вывод этой программы.

```

Datei Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe
rainer@suse:~> greedyGenerator
-10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
rainer@suse:~> █

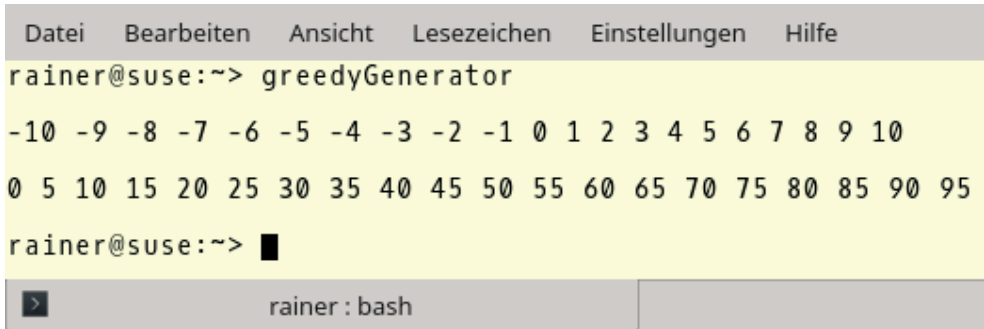
```

Функция-генератор

<sup>1</sup> <http://en.cppreference.com/w/cpp/algorithm/iota>.

Очень важны два наблюдения относительно программы `greedyGenerator.cpp`. С одной стороны, вектор `numbers` в строке 8 всегда содержит все значения. Это справедливо, даже если мне нужно всего первые пять элементов из вектора с 1000 элементов. С другой стороны – очень легко преобразовать `getNumbers` в отложенный (*lazy*) генератор. Следующая программа сознательно оставлена незавершенной. По-прежнему пропущено определение `generator`.

Отложенная функция-генератор



```

Datei  Bearbeiten  Ansicht  Lesezeichen  Einstellungen  Hilfe
rainer@suse:~> greedyGenerator
-10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95
rainer@suse:~> █
rainer : bash

```

В то время как функция `getNumbers` в `greedyGenerator.cpp` возвращала `std::vector<int>`, сопрограмма `generatorForNumbers` в `lazyGenerator.cpp` возвращает генератор. Генератор `numbers` в строке 17 или `generatorForNumbers(0, 5)` в строке 23 возвращает новое число по запросу. Цикл порождает данный запрос. Точнее, запрос сопрограммы возвращает значение `i` через `co_yield i` и немедленно

приостанавливает выполнение. Если запрашивается новое значение, сопрограмма продолжает свое выполнение точно с этого места.

Выражение `generatorForNumbers(0, 5)` в строке 23 – это просто способ использования генератора.

Я хотел бы особенно подчеркнуть один момент. Сопрограмма `generatorForNumbers` создает бесконечный поток данных, поскольку цикл `for` в строке 8 не имеет условия завершения. Это нормально, если я хочу получить конечное количество значений, как, например, в строке 20. Это не справедливо для строки 23, поскольку там такого условия нет. Поэтому это выражение будет выполняться бесконечно.

## 6.1.2 Характеристики

У сопрограмм есть несколько уникальных характеристик.

### 6.1.2.1 Типичные случаи использования

Сопрограммы – это стандартный способ разработки программ, управляемый событиями (*event-driven applications*)<sup>1</sup>. Такими программами могут быть симуляторы, игры, серверы, пользовательский интерфейс или даже алгоритмы. Сопрограммы обычно используются для кооперативной многозадачности (*cooperative multitasking*)<sup>2</sup>. Ключевым в кооперативной многозадачности является то, что каждая задача берет столько времени, сколько ей нужно, но при этом не спит и не ожидает, позволяя вместо этого выполняться другим задачам. Кооперативная многозадачность отличается от вытесняющей многозадачности, в которой должен быть планировщик, определяющий, сколько времени CPU получит каждая задача.

Есть различные типы сопрограмм.

### 6.1.2.2 Базовые понятия

Сопрограммы в C++20 являются асимметричными (*asymmetric*), объектами первого вида (*first class*) и бесстековыми (*stackless*).

Выполнение **асимметричной** сопрограммы возвращается вызывающей стороне. Это неверно для симметричной сопрограммы. Симметричная сопрограмма может передать управление другой сопрограмме.

**Объекты первого вида**, т. е. сопрограммы, похожи на функции первого вида, поскольку сопрограммы ведут себя как данные. Поведение как данные означает, что их можно использовать как аргументы функций или возвращаемые из функций значения, а также запоминать их в переменных.

**Бесстековая** сопрограмма может приостанавливать и возобновлять сопрограмму верхнего уровня. При выполнении сопрограммы управление передается (`yield`) вызывающей стороне. Сопрограмма запоминает свое состояние для возобновления отдельно от стека. Бесстековые сопрограммы часто называют возобновляемыми функциями.

<sup>1</sup> [https://en.wikipedia.org/wiki/Event-driven\\_programming](https://en.wikipedia.org/wiki/Event-driven_programming).

<sup>2</sup> [https://en.wikipedia.org/wiki/Computer\\_multitasking](https://en.wikipedia.org/wiki/Computer_multitasking).

### 6.1.2.3 Цели разработки сопрограмм

Гор Нишанов описывает в предложении N4402<sup>1</sup> цели разработки сопрограмм.

Сопрограммы должны:

- быть сильно масштабируемыми (до миллиардов параллельно выполняемых сопрограмм);
- иметь высокоэффективные операции приостановки и продолжения выполнения, сравнимые по цене с ценой вызова функции;
- легко интегрироваться с существующим функционалом без дополнительных затрат;
- иметь открытое устройство, позволяющее разработчикам библиотек создавать библиотеки сопрограмм, предоставляющие высокоуровневую семантику, такую как генераторы, горутины<sup>2</sup>, задачи и т. п.;
- использоваться в средах, где исключения запрещены или недоступны.

В связи с требованиями масштабируемости и интеграции с существующими возможностями сопрограммы бесстековые. В отличие от них, стековые сопрограммы требуют по умолчанию стек в 1 Мб под Windows и 2 Мб под Linux.

Есть четыре пути, с помощью которых функция может стать сопрограммой.

### 6.1.2.4 Как функция может стать сопрограммой

Функция становится сопрограммой, если она использует:

- `co_return`, или
- `co_await`, или
- `co_yield`, или
- `co_wait`-выражение в цикле на основе диапазона (range-based for).



#### Различайте фабрику сопрограмм и объект-сопрограмму

Сам термин «сопрограмма» часто используется в двух различных аспектах сопрограмм: функция, вызывающая `co_return`, `co_await` или `co_yield`, и сам объект сопрограммы. Использование одного и того же названия для двух разных аспектов может запутать вас (как в свое время путало меня). Давайте проясним оба этих термина.

Простая сопрограмма, возвращающая значение 2021

---

```
MyFuture<int> createFuture() {
    co_return 2021;
}

int main() {

    auto fut = createFuture();
    std::cout << "fut.get(): " << fut.get() << '\n';

}
```

---

<sup>1</sup> <https://isocpp.org/files/papers/N4402.pdf>.

<sup>2</sup> <https://tour.golang.org/concurrency/1>.



В этом примере есть функция `createFuture`, возвращающая объект типа `MyFuture<int>`. И функция, и возвращенный ею объект называются сопрограммами. Если точнее, то функция `createFuture` – это фабрика сопрограмм, которая возвращает объект-сопрограмму. Объект-сопрограма – это возобновляемый (resumable) объект, моделирующий заданное поведение. В разделе о `co_return` я покажу реализацию и использование этой сопрограммы.

#### 6.1.2.4.1 Ограничения

Сопрограммы не могут содержать оператор `return` или заменитель типа для возвращаемых значений. Это относится к `auto` и к концептам.

Кроме того, функции, имеющие переменное количество шаблонных аргументов (variadic templates)<sup>1</sup>, `constexpr` или `constexpr`-функции, конструкторы, деструкторы и функция `main` также не могут быть сопрограммами.

### 6.1.3 Фреймворк

Фреймворк (framework) для реализации сопрограмм состоит из более чем 20 функций, некоторые из которых вы должны определить и некоторые из которых можете переопределить. Тем самым вы специализируете сопрограммы под свои цели.

Сопрограмма ассоциируется с тремя частями: объект-обещание (promise object), дескриптор сопрограммы (coroutine handle) и кадр сопрограммы (coroutine frame). Клиент получает дескриптор сопрограммы для взаимодействия с объектом-обещанием, который хранит свое состояние в кадре сопрограммы.

#### 6.1.3.1 Объект-обещание

Объект-обещание (promise object) управляется изнутри сопрограммы. Сопрограмма передает результат или исключение через этот объект.

Объект-обещание должен поддерживать следующий интерфейс:

| Метод                                 | Описание                                                                           |
|---------------------------------------|------------------------------------------------------------------------------------|
| Конструктор по умолчанию              | Объект-обещание должен иметь конструктор по умолчанию                              |
| <code>initial_suspend()</code>        | Определяет, является ли сопрограма приостановленной (suspend) до своего выполнения |
| <code>final_suspend noexcept()</code> | Определяет, приостанавливается ли сопрограма перед завершением                     |
| <code>unhandled_exception()</code>    | Вызывается, когда происходит исключение                                            |
| <code>get_return_object()</code>      | Возвращает объект-сопрограму                                                       |
| <code>return_value(val)</code>        | Вызывается <code>co_return val</code>                                              |
| <code>return_void()</code>            | Вызывается <code>co_return</code>                                                  |
| <code>yield_value(val)</code>         | Вызывается <code>co_yield val</code>                                               |

<sup>1</sup> [https://en.cppreference.com/w/cpp/language/variadic\\_arguments](https://en.cppreference.com/w/cpp/language/variadic_arguments).

Компилятор автоматически вызывает эти функции во время выполнения сопрограммы. Далее в разделе про исполняемый поток (workflow) мы рассмотрим это подробно.

Функция `get_return_object` возвращает возобновляемый (resumable) объект, который клиент использует для взаимодействия с сопрограммой. Объект-обещание требует по крайней мере один из методов – `return_value`, `return_void` или `yield_value`. Вам не нужно определять методы `return_value` и `return_void`, если ваша сопрограмма никогда не завершает свое выполнение.

Три функции – `yield_value`, `initial_suspend` и `final_suspend` – возвращают ожидаемые объекты (awaitable). Ожидаемый объект – это то, чего вы можете ожидать. Ожидаемый объект определяет, приостанавливается сопрограмма или нет.

### 6.1.3.2 Хендлер сопрограммы

Хендлер сопрограммы (handle, дескриптор) – это невладеющий дескриптор для продолжения выполнения или уничтожения фрейма сопрограммы снаружи. Дескриптор сопрограммы – это часть возобновляемой функции.

Следующий фрагмент кода – простой `Generator`, содержащий дескриптор сопрограммы `coro`.

Дескриптор сопрограммы

---

```
1  template<typename T>
2  struct Generator {
3
4      struct promise_type;
5      using handle_type = std::coroutine_handle<promise_type>;
6
7      Generator(handle_type h): coro(h) {}
8      handle_type coro;
9
10     ~Generator() {
11         if ( coro ) coro.destroy();
12     }
13     T getValue() {
14         return coro.promise().current_value;
15     }
16     bool next() {
17         coro.resume();
18         return not coro.done();
19     }
20     ...
21 }
```

---

Конструктор (строка 7) получает дескриптор сопрогаммы, который имеет тип `std::coroutine_handle<promise_type>`<sup>1</sup>. Методы `next` (строка 16) и `getValue` (строка 13) позволяют клиенту возобновить выполнение объекта-обещания (`gen.next()`) или запросить у него его значение (`gen.getValue()`) при помощи дескриптора сопрогаммы.

Вызов сопрогаммы

---

```
Generator<int> coroutineFactory(); // function that return
                                // a coroutine object

auto gen = coroutineFactory();
gen.next();
auto result = gen.getValue();
```

---

Внутри оба метода используют дескриптор сопрогаммы `coro` (строка 8) для:

- возобновления выполнения сопрогаммы: `coro.resume()` (строка 17) или `coro()`;
- уничтожения сопрогаммы `coro.destroy()` (строка 11);
- проверки состояния сопрогаммы: `coro` (строка 11).

Сопрограмма автоматически уничтожается, когда ее тело завершает свою работу. Вызов `coro` возвращает `true` только в ее последней точке приостановки.



#### **Возобновляемый объект требует использования внутреннего типа `promise_type`**

Возобновляемый объект, такой как `Generator`, должен иметь внутренний тип `promise_type`. Или вы можете специализировать `std::coroutine_traits`<sup>2</sup> для `Generator` и определить публичный тип `promise_type` в `std::coroutine_traits<Generator>`.

### **6.1.3.3 Фрейм сопрогаммы**

Фрейм сопрогаммы (`coroutine frame`) – это внутреннее, обычно выделенное на стеке состояние. Оно состоит из уже упомянутого объекта-обещания, скопированных параметров сопрогаммы, представления точек приостановки (`suspension point`), локальных переменных, чье время жизни заканчивается перед текущей точкой приостановки, и локальных переменных, чье время жизни продолжается после текущей точки приостановки.

Для оптимизации выделения памяти под сопрограмму необходимо выполнение двух требований:

- 1) время жизни сопрогаммы должно быть вложено во время жизни вызывающего объекта;
- 2) вызывающий сопрограмму объект должен знать размер кадра сопрогаммы.

Наиболее важными абстракциями в библиотеке сопрограмм являются ожидаемые объекты (`awaitable`) и ожидающие объекты (`awaiter`).

<sup>1</sup> [https://en.cppreference.com/w/cpp/coroutine/coroutine\\_handle](https://en.cppreference.com/w/cpp/coroutine/coroutine_handle).

<sup>2</sup> [https://en.cppreference.com/w/cpp/coroutine/coroutine\\_traits](https://en.cppreference.com/w/cpp/coroutine/coroutine_traits).

## 6.1.4 Ожидаемые и ожидающие объекты

Три метода объекта-обещания – `yield_value`, `initial_suspend` и `final_suspend` – возвращают ожидаемый объект.

### 6.1.4.1 Ожидаемые объекты

Ожидаемый объект (awaitable) – это что-то, что вы можете ожидать. Такой объект определяет, приостанавливается сопрограмма или нет.

Используя объект-ожидание `prom` и оператор `co_await`, компилятор генерирует три вызова функций.

| Вызов                               | Генерируемый компилятором вызов               |
|-------------------------------------|-----------------------------------------------|
| <code>yield value</code>            | <code>co_await prom.yield_value(value)</code> |
| <code>prom.initial_suspend()</code> | <code>co_await prom.initial_suspend()</code>  |
| <code>prom.final_suspend()</code>   | <code>co_await prom.final_suspend()</code>    |

Оператор `co_await` требует ожидаемый объект как аргумент. Ожидаемые объекты должны реализовывать концепт `Awaitable`.

### 6.1.4.2 Концепт `Awaitable`

Концепт `Awaitable` требует реализации трех методов.

| Метод                      | Описание                                                                                                       |
|----------------------------|----------------------------------------------------------------------------------------------------------------|
| <code>await_ready</code>   | Обозначает, готов ли результат. Когда возвращает <code>false</code> , то вызывается <code>await_suspend</code> |
| <code>await_suspend</code> | Управляет (schedule) сопрограммой: выполнить операцию возобновления или уничтожения                            |
| <code>await_resume</code>  | Возвращает результат для выражения <code>co_await exp</code>                                                   |

В стандарте C++20 уже определены два базовых ожидаемых объекта: `std::suspend_always` и `std::suspend_never`.

### 6.1.4.3 `std::suspend_always` И `std::suspend_never`

Как подсказывает его название, объект `suspend_always` всегда приостанавливает выполнение. Поэтому вызов `await_ready` возвращает `false`.

Ожидаемый объект `std::suspend_always`

---

```
struct suspend_always {
    constexpr bool await_ready() const noexcept { return false; }
    constexpr void await_suspend(std::coroutine_handle<>) const noexcept {}
    constexpr void await_resume() const noexcept {}
};
```

---

Противоположное справедливо для `std::suspend_never`. Он никогда не приостанавливает выполнение, и вызов `await_ready` всегда возвращает `true`.

Ожидаемый объект `std::suspend_never`

---

```
struct suspend_never {
    constexpr bool await_ready() const noexcept { return true; }
    constexpr void await_suspend(std::coroutine_handle<>) const noexcept {}
    constexpr void await_resume() const noexcept {}
};
```

---

Ожидаемые объекты `std::suspend_always` и `std::suspend_never` – это базовые строительные блоки для функций, таких как `initial_suspend` и `final_suspend`. Обе функции автоматически выполняются, когда осуществляется выход из сопрограммы: `initial_suspend` в начале и `final_suspend` в конце сопрограммы.

#### 6.1.4.4 `initial_suspend`

Когда метод `initial_suspend` возвращает `std::suspend_always`, то сопрограмма приостанавливается в самом начале. При возвращении `std::suspend_never` сопрограмма не приостанавливается.

- Отложенная (lazy) сопрограмма, которая приостанавливается немедленно.

Отложенная сопрограмма

---

```
std::suspend_always initial_suspend() {
    return {};
}
```

---

- Неотложенная (eager) сопрограмма, которая выполняется немедленно.

Неотложенная сопрограмма

---

```
std::suspend_never initial_suspend() {
    return {};
}
```

---

#### 6.1.4.5 `final_suspend`

Когда метод `final_suspend` возвращает `std::suspend_always`, сопрограмма приостанавливается в конце. При возвращении `std::suspend_never` сопрограмма не приостанавливается.

- Отложенная сопрограмма приостанавливается в конце.

Отложенная сопрограмма, которая приостанавливается в конце своего выполнения

---

```
std::suspend_always final_suspend noexcept noexcept noexcept noexcept() {
    return {};
}
```

---

- Неотложенная сопрограмма, которая не приостанавливается в конце своего выполнения

---

```
std::suspend_never final_suspend() noexcept {
    return {};
}
```

---

До сих пор у нас были только ожидаемые (awaitable) объекты. Еще нужен такой объект, который будет ожидать какого-то события. Сейчас я заполню этот пробел и расскажу об ожидающих объектах.

#### 6.1.4.6 Ожидающие объекты

Есть два различных способа получения ожидающего (awaiter) объекта:

- оператор `co_await` определен;
- ожидаемый объект становится ожидающим.

Когда вызывается `co_await expression`, то выражение `expression` является ожидаемым. Само выражение `expression` – это вызов объекта-обещания (ожидаемого): `prom.yield_value(value)`, `prom.initial_suspend()` или `prom.final_suspend()`. Для читаемости я переименовал в следующих строках объект-обещание из `prom` в `awaitable`.

Далее компилятор выполняет следующий алгоритм для получения ожидающего объекта:

1. Он ищет оператор `co_await` для объекта-обещания и возвращает ожидающий объект:  
`awaiter = awaitable.operator co_await();`
2. Он ищет самостоятельный оператор `co_wait` и возвращает ожидающий объект:  
`awaiter = operator co_await();`
3. Если нет определенного оператора `co_wait`, то ожидаемый объект становится ожидающим:  
`awaiter = awaitable;`



#### **awaiter = awaitable**

Когда вы изучите мою реализацию сопрограмм в этой главе, то заметите, что я использую почти все время случай, когда ожидаемый объект неявно становится ожидающим. Только пример в разделе по синхронизации потоков использует оператор `co_await` для получения ожидающего объекта.

После всех этих статических аспектов сопрограмм я хочу перейти к их динамическим аспектам.

### 6.1.5 Исполняемый поток процессов

Компилятор преобразует вашу сопрограмму и выполняет два исполняемых потока процессов (workflow): внешний процесс объекта-обещания и внутренний процесс ожидающего.

### 6.1.5.1 Процесс объекта-ожидания

Когда вы используете `co_yield`, `co_await` или `co_return` в функции, то функция становится сопрограммой, и компилятор преобразует ее тело во что-то, эквивалентное следующим строкам кода:

Преобразованная сопрограмма

---

```

1  {
2      Promise prom;
3      co_await prom.initial_suspend();
4      try {
5          <function body having co_return, co_yield, or co_wait>
6      }
7      catch (...) {
8          prom.unhandled_exception();
9      }
10 FinalSuspend:
11     co_await prom.final_suspend();
12 }
```

---

Компилятор автоматически выполняет преобразованный код, используя методы объекта-обещания. Я называю такой процесс процессом объекта-обещания. Вот его основные этапы.

- Сопрограмма начинает выполнение:
  - ♦ выделяет фрейм сопрограммы, если необходимо;
  - ♦ копирует все параметры во фрейм сопрограммы;
  - ♦ создает объект `prom` (строка 2);
  - ♦ вызывает для `prom.get_return_object()` создание дескриптора сопрограммы и сохраняет его в локальной переменной. Результат вызова будет возвращен при первой приостановке сопрограммы;
  - ♦ вызывает `prom.initial_suspend()` и ожидает его результат при помощи `co_await`. Объект-обещание обычно возвращает `suspend_never` для «жадных» (*eagerly-started*) сопрограмм и `suspend_always` для отложенных (*eagerly-started*) сопрограмм (строка 3);
  - ♦ тело сопрограммы выполняется тогда, когда `co_await prom.initial_suspend()` возобновляет выполнение.
- Сопрограмма достигает точки приостановки:
  - ♦ возвращаемый объект (`prom.get_return_object()`) возвращается вызывающей стороне, которая возобновляет сопрограмму.
- Сопрограмма достигает `co_return`:
  - ♦ вызывает `prom.return_void()` для `co_return` или `co_return expression`, где `expression` имеет тип `void`;
  - ♦ вызывает `prom.return_value(expression)` для `co_return expression`, где `expression` имеет тип, отличный от `void`;

- ◆ уничтожает все связанные со стеком переменные;
- ◆ вызывает `prom.final_suspend()` и ожидает его результат при помощи `co_await`.
- Сопрограмма уничтожается (уничтожая через `co_return` непойманное исключение или через дескриптор сопрограммы):
  - ◆ вызывает уничтожение объекта-обещания;
  - ◆ вызывает деструктор параметров функции;
  - ◆ освобождает память, используемую фреймом сопрограммы;
  - ◆ передает управление обратно вызывающей стороне.

Когда сопрограмма завершается с необработанным исключением, то происходит следующее:

- исключение перехватывается, и вызывается `prom.unhandled_exception()` из блока `catch`;
- вызывается `prom.final_suspend()`, и ожидается результат через `co_await` (строка 11).

Когда происходит использование `co_await expr` в сопрограмме или компилятор неявно вызывает `co_await prom.initial_suspend()`, `co_await prom.final_suspend()` или `co_await prom.yield_value(value)`, то второй внутренний процесс – процесс ожидающего объекта – запускается.

### 6.1.5.2 Поток ожидающих процессов

Использование `co_await expr` заставляет компилятор преобразовать код, основанный на функциях `await_ready`, `await_suspend` и `await_resume`. Соответственно, я называю выполнение преобразованного кода потоком ожидающих процессов (*awaiter workflow*).

Компилятор генерирует примерно следующий код, использующий `awaitable`. Для простоты я игнорирую обработку исключений и описываю процесс при помощи комментариев.

Созданный поток ожидающих процессов

---

```
1  awaitable.await_ready() returns false:
2
3  suspend coroutine
4
5  awaitable.await_suspend(coroutineHandle) returns:
6
7  void:
8      awaitable.await_suspend(coroutineHandle);
9      coroutine keeps suspended
10     return to caller
11
12  bool:
13     bool result = awaitable.await_suspend(coroutineHandle);
14     if result:
15         coroutine keep suspended
16     return to caller
```



```

17     else:
18         go to resumptionPoint
19
20     another coroutine handle:
21         auto anotherCoroutineHandle = awaitable.await_suspend(coroutineHandle);
22         anotherCoroutineHandle.resume();
23         return to caller
24
25 resumptionPoint:
26
27 return awaitable.await_resume();

```

Процесс выполняется, только если `awaitable.await_ready()` возвращает `false` (строка 1). В случае когда возвращается `true`, сопрограмма готова и возвращает результат вызова `awaitable.await_resume()` (строка 27).

Давайте предположим, что `awaitable.await_ready()` возвращает `false`. Тогда сопрограмма приостанавливается (строка 3) и немедленно обрабатывает значение вызова `awaitable.await_ready()`. Возвращаемым типом может быть `void` (строка 7), `bool` (строка 12) или же дескриптор другой сопрограммы `anotherCoroutineHandle` (строка 20). В зависимости от типа возвращаемого значения происходит возврат управления или же выполняется другая сопрограмма.

Возвращенное значение вызова `awaitable.await_suspend()`

| Тип                                 | Описание                                                                                                                                                                                                              |
|-------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>void</code>                   | Сопрограмма остается приостановленной, и управление возвращается вызывающей стороне                                                                                                                                   |
| <code>bool</code>                   | <code>bool == true</code> сопрограмма остается приостановленной, и управление возвращается вызывающей стороне<br><code>bool == false</code> возобновляется выполнение сопрограммы, и не происходит возврат управления |
| <code>anotherCoroutineHandle</code> | Возобновляется выполнение другой сопрограммы, и потом происходит возврат управления вызывающей стороне                                                                                                                |

Что происходит, когда происходит исключение? Важно, где оно происходит – в `await_read`, `await_suspend` или `await_resume`.

- `await_ready`: сопрограмма не приостанавливается, и не выполняются вызовы `await_suspend` и `await_resume`.
- `await_suspend`: исключение перехватывается, возобновляется выполнение сопрограммы, и исключение вызывается повторно. `await_resume` не вызывается.
- `await_resume`: выполняются `await_ready` и `await_suspend`, и все значения возвращаются. Вызов `await_resume` не возвращает результат.

Давайте теперь применим теорию к практике.

## 6.1.6 co\_return

Сопрограмма использует `co_return` в качестве своего оператора возврата результата работы.

### 6.1.6.1 Future

Сопрограмма, приведенная в программе `eagerFuture.cpp`, – это простейшая сопрограмма, которую я только могу себе представить. При этом она делает полезную работу: автоматически запоминает результат своего вызова.

Неотложенный объект

---

```
1  // eagerFuture.cpp
2
3  #include <coroutine>
4  #include <iostream>
5  #include <memory>
6
7  template<typename T>
8  struct MyFuture {
9      std::shared_ptr<T> value;
10     MyFuture(std::shared_ptr<T> p): value(p) {}
11     ~MyFuture() { }
12     T get() {
13         return *value;
14     }
15
16     struct promise_type {
17         std::shared_ptr<T> ptr = std::make_shared<T>();
18         ~promise_type() { }
19         MyFuture<T> get_return_object() {
20             return ptr;
21         }
22         void return_value(T v) {
23             *ptr = v;
24         }
25         std::suspend_never initial_suspend() {
26             return {};
27         }
28         std::suspend_never final_suspend() noexcept {
29             return {};
30         }
31     }
```

```

31         void unhandled_exception() {
32             std::exit(1);
33         }
34     };
35 };
36
37 MyFuture<int> createFuture() {
38     co_return 2021;
39 }
40
41 int main() {
42
43     std::cout << '\n';
44
45     auto fut = createFuture();
46     std::cout << "fut.get(): " << fut.get() << '\n';
47
48     std::cout << '\n';
49
50 }

```

Класс `MyFuture` ведет себя как объект `future`<sup>1</sup>, который выполняется немедленно. Вызов сопрограммы `createFuture` (строка 45) возвращает объект `future`, и вызов `fut.get()` (строка 46) берет значение связанного объекта-обещания.

Но есть одна особенность для объекта `future`: возвращенное значение сопрограммы `createFuture` доступно после вызова. Благодаря особенностям времени жизни возвращаемое значение управляется при помощи `std::shared_ptr` (строки 9 и 17). Сопрограмма всегда использует `std::suspend_never` (строки 25 и 28) и поэтому ни до, ни после выполнения не приостанавливает своего выполнения. Это значит, что сопрограмма выполняется тогда, когда вызывается `createFuture`. Метод `get_return_object` (строка 19) создает и запоминает дескриптор сопрограммы, и `return_value` (строка 22) запоминает результат работы сопрограммы, который был возвращен строчкой `co_return 2021` (строка 38). Клиент вызывает `fut.get` (строка 46) и использует объект `future` как дескриптор объекта-обещания. Метод `get` возвращает результат клиенту (строка 13).

```
fut.get() : 2021
```

Неотложенный объект `future`

<sup>1</sup> <https://en.cppreference.com/w/cpp/thread/future>.

Можно подумать, что вряд ли стоило показывать реализацию сопрограммы, которая ведет себя как функция. Но эта простая сопрограмма – идеальная отправная точка для разработки различных реализаций объектов `future`. Об этом рассказывается в разделе «Варианты `future`» далее в книге.

## 6.1.7 `co_yield`

Благодаря `co_yield` вы можете реализовать генератор, создающий бесконечный поток данных, из которого вы можете брать значения. Возвращаемый тип генератора `generatorForNumbers(int begin, int inc=1)` – это `generator<int>`, при этом сам генератор содержит внутри себя специальный объект-обещание `p`, такой что вызов `co_yield i` эквивалентен вызову `co_await p.yield_value(i)`. Оператор `co_yield i` может быть вызван произвольное число раз. Сразу после каждого вызова выполнение сопрограммы приостанавливается.

### 6.1.7.1 Бесконечный поток данных

Программа `infiniteDataStream.cpp` создает бесконечный поток данных (`infinite data stream`). Сопрограмма `getNext` использует `co_yield` для того, чтобы создать поток данных, который начинается со `start` и при получении очередного запроса увеличивает значение на `step`.

Бесконечный поток данных

---

```
1  // infiniteDataStream.cpp
2
3  #include <coroutine>
4  #include <memory>
5  #include <iostream>
6
7  template<typename T>
8  struct Generator {
9
10     struct promise_type;
11     using handle_type = std::coroutine_handle<promise_type>;
12
13     Generator(handle_type h): coro(h) {} // (3)
14     handle_type coro;
15
16     ~Generator() {
17         if ( coro ) coro.destroy();
18     }
19     Generator(const Generator&) = delete;
20     Generator& operator = (const Generator&) = delete;
```

```

21     Generator(Generator&& oth) noexcept : coro(oth.coro) {
22         oth.coro = nullptr;
23     }
24     Generator& operator = (Generator&& oth) noexcept {
25         coro = oth.coro;
26         oth.coro = nullptr;
27         return *this;
28     }
29     T getValue() {
30         return coro.promise().current_value;
31     }
32     bool next() {                                     // (5)
33         coro.resume();
34         return not coro.done();
35     }
36     struct promise_type {
37         promise_type() = default;                     // (1)
38
39         ~promise_type() = default;
40
41         auto initial_suspend() {                       // (4)
42             return std::suspend_always{};
43         }
44         auto final_suspend() noexcept {
45             return std::suspend_always{};
46         }
47         auto get_return_object() {                     // (2)
48             return Generator{handle_type::from_promise(*this)};
49         }
50         auto return_void() {
51             return std::suspend_never{};
52         }
53
54         auto yield_value(const T value) {              // (6)
55             current_value = value;
56             return std::suspend_always{};
57         }

```

```
58         void unhandled_exception() {
59             std::exit(1);
60         }
61         T current_value;
62     };
63
64 };
65
66 Generator<int> getNext(int start = 0, int step = 1) {
67     auto value = start;
68     while (true) {
69         co_yield value;
70         value += step;
71     }
72 }
73
74 int main() {
75
76     std::cout << '\n';
77
78     std::cout << "getNext():";
79     auto gen = getNext();
80     for (int i = 0; i <= 10; ++i) {
81         gen.next();
82         std::cout << " " << gen.getValue();    // (7)
83     }
84
85     std::cout << "\n\n";
86
87     std::cout << "getNext(100, -10):";
88     auto gen2 = getNext(100, -10);
89     for (int i = 0; i <= 20; ++i) {
90         gen2.next();
91         std::cout << " " << gen2.getValue();
92     }
93
94     std::cout << '\n';
95
96 }
```

---

Программа `main` создает две сопрограммы. Первая сопрограмма `gen` (строка 79) возвращает значения от 0 до 10, вторая сопрограмма `gen2` (строка 88) возвращает значения от 100 до -100. Прежде чем я более детально опишу процесс работы программы благодаря онлайн-компилятору Wandbox<sup>1</sup>, изучите вывод этой программы.

```
Start

getNext(): 0 1 2 3 4 5 6 7 8 9 10

getNext(100, -10): 100 90 80 70 60 50 40 30 20 10 0 -10 -20 -30 -40 -50 -60 -70 -80 -90 -100

0

Finish
```

Бесконечный поток данных

Числа в программе `infiniteDataStream.cpp` обозначают шаги в первой итерации потока исполняемых процессов:

1. Создаем объект-обещание.
2. Вызываем `promise.get_return_object()` и сохраняем результат в локальной переменной.
3. Создаем генератор.
4. Вызываем `promise.initial_suspend()`. Генератор отложенный (*lazy*) и поэтому всегда приостанавливает свое выполнение.
5. Запрашивает следующее значение и возвращает его.
6. Запускается вызовом `co_yield`. После этого доступно следующее значение.
7. Получает следующее значение.

В дополнительных итерациях выполняются только шаги 5, 6 и 7.

В разделе «Изменение и обобщение потоков» далее в книге обсуждаются различные улучшения и изменения генератора `infiniteDataStream.cpp`.

### 6.1.8 `co_await`

`co_await` вызывает приостановку или продолжение выполнения сопрограммы. Выражение `expr` в `co_await expr` должно быть ожидаемым выражением (*awaitable expression*), т. е. должно реализовывать конкретный интерфейс, состоящий из трех методов: `await_ready`, `await_suspend` и `await_resume`.

Типичным примером использования `co_await` является сервер, ожидающий событий.

<sup>1</sup> <https://wandbox.org/>.

Блокирующий сервер

---

```
1  Acceptor acceptor{443};
2  while (true) {
3      Socket socket = acceptor.accept();           // blocking
4      auto request = socket.read();               // blocking
5      auto response = handleRequest(request);
6      socket.write(response);                     // blocking
7  }
```

---

Сервер довольно прост, поскольку он по очереди отвечает на каждый запрос в одном и том же потоке. Сервер слушает порт 443 (строка 1), подтверждает соединение (строка 3), читает входные данные от клиента (строка 4) и посылает ответ клиенту (строка 6). Вызовы в строках 3, 4 и 6 являются блокирующими.

Благодаря `co_await` блокирующие вызовы могут быть приостановлены и возобновлены.

Ожидающий сервер

---

```
1  Acceptor acceptor{443};
2  while (true) {
3      Socket socket = co_await acceptor.accept();
4      auto request = co_await socket.read();
5      auto response = handleRequest(request);
6      co_await socket.write(response);
7  }
```

---

Прежде чем я перейду к действительно интересному примеру с синхронизацией потоков при помощи сопрограмм, я хотел бы начать с чего-то, достаточно простого: начала нового задания (`job`) по запросу.

#### 6.1.8.1 Создание задания по запросу

Сопрограмма в следующем примере настолько проста, насколько это возможно. Она ожидает предопределенного ожидаемого объекта `std::suspend_never()`.

Создание задания по запросу

---

```
1  // startJob.cpp
2
3  #include <coroutine>
4  #include <iostream>
5
```



```

6  struct Job {
7      struct promise_type;
8      using handle_type = std::coroutine_handle<promise_type>;
9      handle_type coro;
10     Job(handle_type h): coro(h){}
11     ~Job() {
12         if ( coro ) coro.destroy();
13     }
14     void start() {
15         coro.resume();
16     }
17
18
19     struct promise_type {
20         auto get_return_object() {
21             return Job{handle_type::from_promise(*this)};
22         }
23         std::suspend_always initial_suspend() {
24             std::cout << "    Preparing job" << '\n';
25             return {};
26         }
27         std::suspend_always final_suspend() noexcept {
28             std::cout << "    Performing job" << '\n';
29             return {};
30         }
31         void return_void() {}
32         void unhandled_exception() {}
33
34     };
35 };
36
37 Job prepareJob() {
38     co_await std::suspend_never();
39 }
40
41 int main() {
42
43     std::cout << "Before job" << '\n';
44

```

```
45     auto job = prepareJob();
46     job.start();
47
48     std::cout << "After job" << '\n';
49
50 }
```

---

Вы можете подумать, что сопрограмма `prepareJob` (строка 37) бессмысленна, поскольку ожидаемый объект всегда приостанавливается. Это не так. Функция `prepareJob` – это фабрика сопрограмм, использующая `co_await` (строка 38) и возвращающая объекты-сопрограммы. Функция `prepareJob()` в строке 45 создает сопрограмму типа `Job`. Эта сопрограмма сразу же приостанавливается, поскольку метод объекта-обещания возвращает `std::suspend_always` (строка 23). Именно по этой причине необходим вызов `job.start` (строка 46) для возобновления сопрограммы (строка 15). Метод `final_suspend` также возвращает `std::suspend_always` (строка 27).

```
Before job
    Preparing job
    Performing job
After job
```

Создание задания по запросу

Далее в книге я буду использовать программу `startJob` как базу для дальнейших экспериментов.

#### 6.1.8.2 Синхронизация потоков

Часто потоки сами себя синхронизируют. Один поток подготавливает задание, другой поток его ждет. Условные переменные (condition variables)<sup>1</sup>, обещания и объекты `future`<sup>2</sup>, а также атомарные логические переменные<sup>3</sup> могут быть использованы для организации работы вида получатель–отправитель. Благодаря сопрограммам синхронизация потоков становится крайне не простой, без неизбежных рисков, связанных с возможными событиями ложного пробуждения (spurious wakeup) или потерянного пробуждения (lost wakeup).

---

<sup>1</sup> [https://en.cppreference.com/w/cpp/thread/condition\\_variable](https://en.cppreference.com/w/cpp/thread/condition_variable).

<sup>2</sup> <https://en.cppreference.com/w/cpp/thread>.

<sup>3</sup> <https://en.cppreference.com/w/cpp/atomic/atomic>.

## Синхронизация потоков

---

```

1  // senderReceiver.cpp
2
3  #include <coroutine>
4  #include <chrono>
5  #include <iostream>
6  #include <functional>
7  #include <string>
8  #include <stdexcept>
9  #include <atomic>
10 #include <thread>
11
12 class Event {
13     public:
14
15         Event() = default;
16
17         Event(const Event&) = delete;
18         Event(Event&&) = delete;
19         Event& operator=(const Event&) = delete;
20         Event& operator=(Event&&) = delete;
21
22         class Awaiter;
23         Awaiter operator co_await() const noexcept;
24
25         void notify() noexcept;
26
27     private:
28
29         friend class Awaiter;
30
31         mutable std::atomic<void*> suspendedWaiter{nullptr};
32         mutable std::atomic<bool> notified{false};
33
34 };
35
36 class Event::Awaiter {
37     public:
38         Awaiter(const Event& eve): event(eve) {}
39

```

```
40     bool await_ready() const;
41     bool await_suspend(std::coroutine_handle<> corHandle) noexcept;
42     void await_resume() noexcept {}
43
44 private:
45     friend class Event;
46
47     const Event& event;
48     std::coroutine_handle<> coroutineHandle;
49 };
50
51 bool Event::Awaiter::await_ready() const {
52
53     // allow at most one waiter
54     if (event.suspendedWaiter.load() != nullptr){
55         throw std::runtime_error("More than one waiter is not valid");
56     }
57
58     // event.notified == false; suspends the coroutine
59     // event.notified == true; the coroutine is executed like a normal function
60     return event.notified;
61 }
62
63 bool Event::Awaiter::await_suspend(std::coroutine_handle<>
64                                     corHandle) noexcept {
65     coroutineHandle = corHandle;
66
67     if (event.notified) return false;
68
69     // store the waiter for later notification
70     event.suspendedWaiter.store(this);
71
72     return true;
73 }
74
75 void Event::notify() noexcept {
76     notified = true;
77
78     // try to load the waiter
79     auto* waiter = static_cast<Awaiter*>(suspendedWaiter.load());
80 }
```

```

81     // check if a waiter is available
82     if (waiter != nullptr) {
83         // resume the coroutine => await_resume
84         waiter->coroutineHandle.resume();
85     }
86 }
87
88 Event::Awaiter Event::operator co_await() const noexcept {
89     return Awaiter{ *this };
90 }
91
92 struct Task {
93     struct promise_type {
94         Task get_return_object() { return {}; }
95         std::suspend_never initial_suspend() { return {}; }
96         std::suspend_never final_suspend() noexcept { return {}; }
97         void return_void() {}
98         void unhandled_exception() {}
99     };
100 };
101
102 Task receiver(Event& event) {
103     auto start = std::chrono::high_resolution_clock::now();
104     co_await event;
105     std::cout << "Got the notification! " << '\n';
106     auto end = std::chrono::high_resolution_clock::now();
107     std::chrono::duration<double> elapsed = end - start;
108     std::cout << "Waited " << elapsed.count() << " seconds." << '\n'
109 }
110
111 using namespace std::chrono_literals;
112
113 int main() {
114
115     std::cout << '\n';
116
117     std::cout << "Notification before waiting" << '\n';
118     Event event1{};
119     auto senderThread1 = std::thread([&event1]{ event1.notify(); });
120                                     // Notificatic
121     auto receiverThread1 = std::thread(receiver, std::ref(event1));

```

```
122     receiverThread1.join();
123     senderThread1.join();
124
125     std::cout << '\n';
126
127     std::cout << "Notification after 2 seconds waiting" << '\n';
128     Event event2{};
129     auto receiverThread2 = std::thread(receiver, std::ref(event2));
130     auto senderThread2 = std::thread([&event2]{
131         std::this_thread::sleep_for(2s);
132         event2.notify();
133     });                                     // Notificatio
134
135     receiverThread2.join();
136     senderThread2.join();
137
138     std::cout << '\n';
139
140 }
```

---

С точки зрения пользователя синхронизация потоков с использованием сопрограмм проста. Давайте посмотрим на программу `senderReceiver.cpp`. Каждый из потоков `senderThread1` (строка 119) и `senderThread2` (строка 130) использует события для отправки уведомления в строках 119 и 132. Функция `receiver` в строках 102–109 – это сопрограмма, которая выполняется в потоках `receiverThread1` (строка 122) и `receiverThread2` (строка 135). Я замерил время между началом и концом сопрограммы и вывел его в поток вывода. Оно показывает, как долго сопрограмма ждет. Следующий скриншот демонстрирует вывод программы.

```
Start

Notification before waiting
Got the notification!
Waited 1.5738e-05 seconds.

Notification after 2 seconds waiting
Got the notification!
Waited 2.00019 seconds.

0
Finish
```

Если вы сравните класс `Generator` для генерации бесконечного потока данных с классом `Event` в этом примере, то заметите небольшую разницу. В первом случае класс `Generator` является и ожидаемым, и ожидающим; во втором случае `Event` использует оператор `co_await` для возвращения ожидающего объекта. Это разделение улучшает структуру кода.

Вывод показывает, что выполнение второй сопрограммы занимает две секунды. Причиной является то, что `event1` посылает уведомление (строка 119), прежде чем приостанавливается сопрограмма, но `event2` посылает свое уведомление после интервала времени в две секунды (строка 132).

Теперь рассмотрим программу с позиции разработчика. Работа данной сопрограммы довольно сложна для понимания. У класса `Event` есть два интересных поля: `suspendedWaiter` и `notified`. Переменная `suspendedWaiter` в строке 31 содержит ожидающий сигнала объект, и `notified` содержит состояние оповещения (`notification`).

В своем объяснении обоих процессов я исхожу из того, что в первом случае (первый процесс) уведомление о событии происходит до того, как сопрограмма будет ожидать события. Во втором случае (второй процесс) я предполагаю обратную ситуацию.

Давайте посмотрим на `event1` и первый поток. Здесь `event1` посылает свое уведомление до того, как запускается `receiverThread1`. Вызов `event1` (строка 118) запускает метод `notify` (строки 75–86). Выставляется первый флаг уведомления, и происходит вызов `static_cast<Awaiter*>(suspendedWaiter.load());`. Этот вызов загружает потенциального ожидающего (`waiter`). В этом случае `waiter` равен `nullptr`, поскольку он не был установлен ранее. Это значит, что следующий вызов `resume` для `waiter` в строке 84 не будет выполнен. Вызываемая далее функция `await_ready` (строки 51–61) сначала проверяет, существует ли больше одного `waiter`. В этом случае вызывается исключение `std::runtime`. Главная часть этого метода – это возвращаемое значение. Значение `event.notification` уже установлено как `true` в методе `notify`. Значение `true` означает в этом случае то, что сопрограмма не приостановлена и выполняется как обычная функция.

Во втором потоке вызов `co_await event2` происходит перед тем, как `event2` посылает свое уведомление. Соответственно, `co_await event2` производит вызов `await_ready` (строка 51). Принципиальным отличием от первого потока является то, что `event.notification` равен `false`. Это значение приводит к приостановке выполнения сопрограммы. Формально метод `await_suspend` (строки 63–73) выполняется, `await_suspend` получает дескриптор сопрограммы и запоминает его для будущего вызова в переменной `coroutineHandle` (строка 65). Конечно, дальнейший вызов означает возобновление выполнения. Во-вторых, `waiter` хранится в переменной `suspendedWaiter`. Когда далее `event2.notify` вызывает уведомление, то выполняется метод `notify` (строка 75). Отличием от первого процесса будет то, что условие `waiter != nullptr` выполняется. В результате `waiter` использует `coroutineHandle` для возобновления сопрограммы.



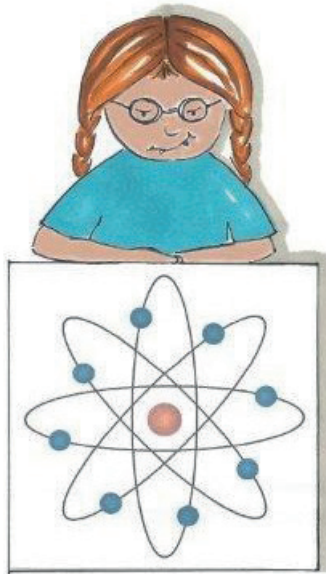
### Краткая информация

- ♦ Сопрограммы (корутины) – это обобщенные функции, которые могут приостанавливать и возобновлять свое выполнение, сохраняя при этом свое состояние.

- ♦ В стандарте C++20 не описаны конкретные сопрограммы, но зато имеется фреймворк для реализации сопрограмм. Он состоит из более чем 20 функций, которые вам надо частично реализовать, а некоторые из них можно перегрузить.
- ♦ При помощи новых ключевых слов `co_await` и `co_yield` в C++20 расширяется выполнение функции в C++ двумя новыми концептами.
- ♦ Благодаря `co_await expression` стало возможным приостанавливать и продолжать выполнение выражения. При использовании `co_await expression` в функции `func` вызов `auto getResult = func()` не блокирует выполнение, если результат функции еще не готов. Вместо потребляющего ресурсы блокирования происходит более дружественное к ресурсам ожидание.
- ♦ `co_yield` позволяет реализовывать бесконечные потоки данных.



## 6.2 Атомарные переменные



Сиппи изучает атомы

Атомарные переменные (атомики, atomics) получили несколько важных расширений в стандарте C++20. Пожалуй, наиболее важным из них стали атомарные ссылки и атомарные умные указатели.

### 6.2.1 `std::atomic_ref`

Шаблонный класс `std::atomic_ref` применяет атомарные операции к объекту, на который он ссылается.

Одновременное чтение и запись атомарного объекта гарантирует, что не будут возникать гонки данных (data race). Время жизни объекта, на который ссылаются, должно превышать время жизни `atomic_ref`. Когда для доступа к объекту используется `std::atomic_ref`, то и все остальные обращения к этому объекту должны идти через `std::atomic_ref`. Кроме того, никакой подобъект объекта, к которому идет обращение через `std::atomic_ref`, не может входить в другой `std::atomic_ref`.

#### 6.2.1.1 Мотивация

Можно подумать, что использование ссылки внутри атомика решает проблему. Но это не так.

В следующей программе я использую класс `ExpensiveToCopy`, который включает в себя `counter`. Поле `counter` одновременно увеличивается с помощью нескольких потоков. Поэтому `counter` должен быть защищен.

Использование атомарной ссылки

---

```
1  // atomicReference.cpp
2
3  #include <atomic>
4  #include <iostream>
5  #include <random>
6  #include <thread>
7  #include <vector>
8
9  struct ExpensiveToCopy {
10     int counter{};
11 };
12
13 int getRandom(int begin, int end) {
14
15     std::random_device seed;           // initial seed
16     std::mt19937 engine(seed());      // generator
17     std::uniform_int_distribution<> uniformDist(begin, end);
18
19     return uniformDist(engine);
20 }
21
22 void count(ExpensiveToCopy& exp) {
23
24     std::vector<std::thread> v;
25     std::atomic<int> counter{exp.counter};
26
27     for (int n = 0; n < 10; ++n) {
28         v.emplace_back([&counter] {
29             auto randomNumber = getRandom(100, 200);
30             for (int i = 0; i < randomNumber; ++i) { ++counter; }
31         });
32     }
33
34     for (auto& t : v) t.join();
35
36 }
37
38 int main() {
39
```

```

40     std::cout << '\n';
41
42     ExpensiveToCopy exp;
43     count(exp);
44     std::cout << "exp.counter: " << exp.counter << '\n';
45
46     std::cout << '\n';
47
48 }

```

Переменная `exp` (строка 42) – это дорогой для копирования объект. Из соображений быстродействия функция `count` (строка 42) получает `exp` по ссылке. Функция `count` инициализирует `std::atomic<int>` через `exp.counter` (строка 25). Следующие строки создают 10 потоков (строка 27), каждый из которых выполняет лямбду, получающую `counter` по ссылке. Лямбда получает случайное число от 100 до 200 (строка 29) и увеличивает счетчик на заданное количество раз. Функция `getRandom` (строка 13) начинает со стартового значения и создает случайное число от 100 до 200 с равномерным законом распределения, используя для этого генератор случайных чисел Вихрь Мерсенна (Mersenne Twister)<sup>1</sup>.

В конце `exp.counter` (строка 44) должна иметь значение примерно 1500, поскольку 10 потоков инкрементировали его в среднем 150 раз. Если выполнить эту программу на [Wandbox online compiler](https://wandbox.org/)<sup>2</sup>, будет получен неожиданный результат.

```

Start
|
| exp.counter: 0
|
| 0
|
Finish

```

Сюрприз с атомарной ссылкой

Счетчик равен 0. Что случилось? Причина проблемы находится в строке 25. Инициализация выражения `std::atomic<int> counter{exp.counter}` создает копию. Следующая небольшая программа иллюстрирует эту проблему.

<sup>1</sup> [https://en.wikipedia.org/wiki/Mersenne\\_Twister](https://en.wikipedia.org/wiki/Mersenne_Twister).

<sup>2</sup> <https://wandbox.org/>.

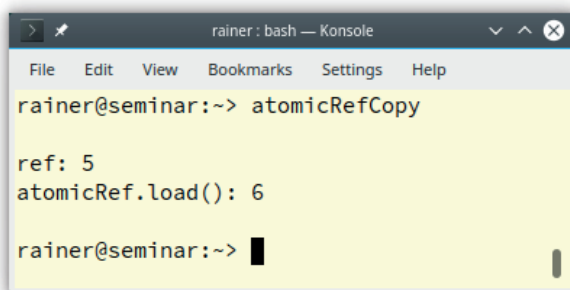
Копирование ссылки

---

```
1 // atomicRefCopy.cpp
2
3 #include <atomic>
4 #include <iostream>
5
6 int main() {
7
8     std::cout << '\n';
9
10    int val{5};
11    int& ref = val;
12    std::atomic<int> atomicRef(ref);
13    ++atomicRef;
14    std::cout << "ref: " << ref << '\n';
15    std::cout << "atomicRef.load(): " << atomicRef.load() << '\n';
16
17    std::cout << '\n';
18
19 }
```

---

Операция инкремента в строке 13 не затрагивает ссылку `ref` (строка 11). Значение `ref` не изменяется.

A screenshot of a terminal window titled "rainer: bash — Konsole". The terminal shows the command "rainer@seminar:~> atomicRefCopy" being executed. The output is "ref: 5" followed by "atomicRef.load(): 6" on the next line. The prompt "rainer@seminar:~>" is visible at the bottom with a cursor. The terminal has a menu bar with "File", "Edit", "View", "Bookmarks", "Settings", and "Help".

```
rainer: bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~> atomicRefCopy

ref: 5
atomicRef.load(): 6

rainer@seminar:~> █
```

## Копирование ссылки

Замена `std::atomic<int>` на `std::atomic_ref<int>` решает проблему.

---

Использование `std::atomic_ref`

---

```
// atomicRef.cpp
```

```
#include <atomic>
#include <iostream>
#include <random>
#include <thread>
#include <vector>

struct ExpensiveToCopy {
    int counter{};
};

int getRandom(int begin, int end) {

    std::random_device seed;           // initial randomness
    std::mt19937 engine(seed());       // generator
    std::uniform_int_distribution<> uniformDist(begin, end);

    return uniformDist(engine);
}

void count(ExpensiveToCopy& exp) {

    std::vector<std::thread> v;
    std::atomic_ref<int> counter{exp.counter};

    for (int n = 0; n < 10; ++n) {
        v.emplace_back([&counter] {
            auto randomNumber = getRandom(100, 200);
            for (int i = 0; i < randomNumber; ++i) { ++counter; }
        });
    }

    for (auto& t : v) t.join();
}
```

```
int main() {  
  
    std::cout << '\n';  
  
    ExpensiveToCopy exp;  
    count(exp);  
    std::cout << "exp.counter: " << exp.counter << '\n';  
  
    std::cout << '\n';  
  
}
```

---

Теперь значение counter такое, как и ожидалось.

```
Start  
|  
|  
| exp.counter: 1531  
|  
| 0  
|  
| Finish
```

Ожидаемый результат с использованием `std::atomic_ref`

Так же, как и `std::atomic`<sup>1</sup>, тип `std::atomic_ref` может иметь специализации и поддерживает специализации для встроенных типов данных.

### 6.2.1.2 Специализации `std::atomic_ref` (C++20)

Вы можете специализировать `std::atomic_ref` для задаваемых пользователем типов, использовать частичные специализации для указателей или полные специализации для арифметических типов, таких как целочисленные или типы с плавающей точкой.

#### 6.2.1.2.1 Базовый шаблон

Базовый шаблон `std::atomic_ref` может быть инстанцирован с любым тривиально копируемым<sup>2</sup> типом T.

---

<sup>1</sup> <https://en.cppreference.com/w/cpp/atomic/atomic>.

<sup>2</sup> [https://en.cppreference.com/w/cpp/types/is\\_trivially\\_copyable](https://en.cppreference.com/w/cpp/types/is_trivially_copyable).

```
struct Counters {
    int a;
    int b;
};
```

```
Counter counter;
std::atomic_ref<Counters> cnt(counter);
```

#### 6.2.1.2.2 Частичные специализации для указателей

Стандарт предоставляет частичные специализации для указателей `std::atomic_ref<T*>`.

#### 6.2.1.2.3 Специализации для арифметических типов

Стандарт предоставляет специализации для целочисленных типов и типов с плавающей точкой: `std::atomic_ref<arithmetic type>`.

- Символьные типы: `char`, `char8_t` (C++ 20), `char16_t`, `char32_t` и `wchar_t`.
- Стандартные знаковые типы: `signed char`, `short`, `int`, `long` и `long long`.
- Стандартные беззнаковые типы: `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long` и `unsigned long long`.
- Дополнительные целые типы, определенные в заголовочном файле `<cstdint>`<sup>1</sup>:
  - ◆ `int8_t`, `int16_t`, `int32_t` и `int64_t` (знаковые целые с 8, 16, 32 и 64 битами);
  - ◆ `uint8_t`, `uint16_t`, `uint32_t` и `uint64_t` (беззнаковые целые с 8, 16, 32 и 64 битами);
  - ◆ `int_fast8_t`, `int_fast16_t`, `int_fast32_t` и `int_fast64_t` (быстрые знаковые типы с не менее 8, 16, 32 и 64 битами);
  - ◆ `uint_fast8_t`, `uint_fast16_t`, `uint_fast32_t` и `uint_fast64_t` (быстрые беззнаковые типы с не менее чем 8, 16, 32 и 64 битами);
  - ◆ `int_least8_t`, `int_least16_t`, `int_least32_t` и `int_least64_t` (наименьшие знаковые целые числа с как минимум 8, 16, 32 и 64 битами);
  - ◆ `uint_least8_t`, `uint_least16_t`, `uint_least32_t` и `uint_least64_t` (наименьшие беззнаковые целые типы с как минимум 8, 16, 32 и 64 битами);
  - ◆ `intmax_t` и `uintmax_t` (максимальные знаковые и беззнаковые целые числа);
  - ◆ `intptr_t` и `uintptr_t` (знаковый и беззнаковый целочисленные типы для хранения указателей);
- стандартные типы с плавающей точкой: `float`, `double` и `long double`.

<sup>1</sup> <http://en.cppreference.com/w/cpp/header/cstdint>.

#### 6.2.1.2.4 Все атомарные операции

Список всех операций, которые можно выполнить над `std::atomic_ref`

| Функция                                               | Описание                                                                                                                                                                                                                                 |
|-------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>is_lock_free</code>                             | Проверяет, является ли объект <code>atomic_ref</code> свободным от блокировки (lock-free)                                                                                                                                                |
| <code>atomic_ref&lt;T&gt;::is_always_lock_free</code> | Проверяет во время компиляции, является ли объект всегда свободным от блокировки                                                                                                                                                         |
| <code>load</code>                                     | Атомарно возвращает значение объекта, на который ссылается                                                                                                                                                                               |
| <code>operator T</code>                               | Атомарно возвращает значение атомика. Эквивалент <code>atom.load()</code>                                                                                                                                                                |
| <code>store</code>                                    | Атомарно заменяет значение объекта, на который ссылается, на неатомарное                                                                                                                                                                 |
| <code>exchange</code>                                 | Атомарно заменяет значение объекта, на который ссылается, на новое                                                                                                                                                                       |
| <code>compare_exchange_strong</code>                  | Атомарно сравнивает и обменивает значение с объектом, на который ссылается                                                                                                                                                               |
| <code>compare_exchange_weak</code>                    | Атомарно сравнивает и обменивает значение объектом, на который ссылается                                                                                                                                                                 |
| <code>fetch_add, +=</code>                            | Атомарно прибавляет значение объекту, на который ссылается                                                                                                                                                                               |
| <code>fetch_sub, -=</code>                            | Атомарно вычитает значение из объекта, на который ссылается                                                                                                                                                                              |
| <code>fetch_or,  =</code>                             | Атомарно выполняет логическое ИЛИ с объектом, на который ссылается                                                                                                                                                                       |
| <code>fetch_and, &amp;=</code>                        | Атомарно выполняет логическое И с объектом, на который ссылается                                                                                                                                                                         |
| <code>fetch_xor, ^=</code>                            | Атомарно выполняет логическое ИСКЛЮЧАЮЩЕЕ ИЛИ с объектом, на который ссылается                                                                                                                                                           |
| <code>++, --</code>                                   | Увеличивает или уменьшает на единицу объект, на который ссылается                                                                                                                                                                        |
| <code>notify_one</code>                               | Разблокирует одну атомарную операцию ожидания                                                                                                                                                                                            |
| <code>notify_all</code>                               | Разблокирует все атомарные операции ожидания                                                                                                                                                                                             |
| <code>wait</code>                                     | Блокирует до уведомления. Сравнивает себя со старым значением для защиты от ложных пробуждений (spurious wakeups) и потерянных пробуждений (lost wakeups). Если значение отличается от старого значения, то просто возвращает управление |



Составные операторы присваивания (`+=`, `-=`, `|=`, `&=` или `^=`) возвращают новое значение, варианты с приставкой `fetch` возвращают старое значение.

Благодаря `constexpr`-функции `atomic_ref<type>::is_lock_free` вы можете проверить каждый атомарный тип, использует ли он блокировки для различных аппаратных конфигураций, на которых может выполняться программа. Эта проверка возвращает `true`, только если она справедлива для всех поддерживаемых аппаратных платформ. Проверка выполняется во время компиляции и доступна начиная с ввода стандарта C++17.

Каждая функция поддерживает дополнительный аргумент, задающий упорядочивание памяти. Значением по умолчанию для этого аргумента является `std::memory_order_seq_cst`, но вы можете также использовать `std::memory_order_relaxed`, `std::memory_order_consume`, `std::memory_order_acquire`, `std::memory_order_release` или `std::memory_order_acq_rel`. Методы `compare_exchange_strong` и `compare_exchange_weak` могут быть параметризованы двумя различными упорядочениями памяти. Одно – для случая успеха, а другое – для случая неудачи. Оба вызова выполняют атомарный обмен (`exchange`) в случае равенства и атомарную загрузку (`load`) в противном случае. Они возвращают `true` в случае успеха и `false` в противном случае. Если вы явно зададите упорядочение памяти, то оно будет использовано для обоих этих случаев. По этой ссылке<sup>1</sup> вы можете прочитать про упорядочение памяти.

Не все операции доступны для всех возможных типов, на которые может ссылаться `std::atomic_ref`. Следующая таблица показывает список всех атомарных операций в зависимости от типа, на который ссылается `std::atomic_ref`.

Все атомарные операции в зависимости от типа, на который ссылается `std::atomic_ref`

| Функция                              | <code>atomic_ref&lt;T&gt;</code> | <code>atomic_ref&lt;floating&gt;</code> | <code>atomic_ref&lt;T*&gt;</code> | <code>atomic_ref&lt;integral&gt;</code> |
|--------------------------------------|----------------------------------|-----------------------------------------|-----------------------------------|-----------------------------------------|
| <code>is_lock_free</code>            | Да                               | Да                                      | Да                                | Да                                      |
| <code>load</code>                    | Да                               | Да                                      | Да                                | Да                                      |
| <code>operator T</code>              | Да                               | Да                                      | Да                                | Да                                      |
| <code>store</code>                   | Да                               | Да                                      | Да                                | Да                                      |
| <code>exchange</code>                | Да                               | Да                                      | Да                                | Да                                      |
| <code>compare_exchange_strong</code> | Да                               | Да                                      | Да                                | Да                                      |
| <code>compare_exchange_weak</code>   | Да                               | Да                                      | Да                                | Да                                      |
| <code>fetch_add, +=</code>           |                                  | Да                                      | Да                                | Да                                      |
| <code>fetch_sub, -=</code>           |                                  | Да                                      | Да                                | Да                                      |
| <code>fetch_or,  =</code>            |                                  |                                         |                                   | Да                                      |
| <code>fetch_and, &amp;=</code>       |                                  |                                         |                                   | Да                                      |

<sup>1</sup> [https://en.cppreference.com/w/cpp/atomic/memory\\_order](https://en.cppreference.com/w/cpp/atomic/memory_order).

Окончание табл.

| Функция                    | <code>atomic_ref&lt;T&gt;</code> | <code>atomic_ref&lt;floating&gt;</code> | <code>atomic_ref&lt;T*&gt;</code> | <code>atomic_ref&lt;integral&gt;</code> |
|----------------------------|----------------------------------|-----------------------------------------|-----------------------------------|-----------------------------------------|
| <code>fetch_xor, ^=</code> |                                  |                                         |                                   | Да                                      |
| <code>++, --</code>        |                                  |                                         | Да                                | Да                                      |
| <code>notify_one</code>    | Да                               | Да                                      | Да                                | Да                                      |
| <code>notify_all</code>    | Да                               | Да                                      | Да                                | Да                                      |
| <code>wait</code>          | Да                               | Да                                      | Да                                | да                                      |

## 6.2.2 Атомарный умный указатель

Объект типа `std::shared_ptr`<sup>1</sup> состоит из контрольного блока и своего ресурса. Контрольный блок потокобезопасен, а вот доступ к ресурсу – нет. Это подразумевает, что изменение счетчика ссылок является атомарной операцией и имеется гарантия того, что ресурс будет удален ровно один раз. Именно эти гарантии и дает `std::shared_ptr`.



### Важность потокобезопасности

Я хотел бы отвлечься от основной темы, чтобы показать, насколько важно то, что `std::shared_ptr` обладает четко определенной многопоточной семантикой. На первый взгляд, использование `std::shared_ptr` не кажется правильным выбором для многопоточного кода. Он по определению общий и изменяемый и является идеальным кандидатом для несинхронизированных операций чтения и записи, а поэтому предназначен для неопределенного поведения. С другой стороны, есть рекомендации в современном C++ **не использовать необработанные указатели** (raw pointers). Это значит, что вы должны использовать умные указатели (smart pointers) в многопоточных программах.

Предложение N1462<sup>2</sup> по атомарным умным указателям непосредственно описывает недостатки текущей реализации. Эти недостатки сводятся к следующим трем пунктам: консистентность, правильность и быстродействие.

- **Консистентность:** атомарные операции для `std::shared_ptr` являются единственными атомарными операциями для неатомарного типа.
- **Правильность:** использование глобальных атомарных операций подвержено ошибкам, поскольку правильное применение держится на дисциплине. Легко забыть использовать атомарную операцию, такую как `ptr = localPtr`, вместо `std::atomic_store(&ptr, localPtr)`. Результатом будет неопределенное поведение (undefined behavior) из-за состояния гонки (data race). Если вместо этого мы используем атомарный умный указатель, то система типов этого просто не позволит.

<sup>1</sup> [https://en.cppreference.com/w/cpp/memory/shared\\_ptr](https://en.cppreference.com/w/cpp/memory/shared_ptr).

<sup>2</sup> <http://wg21.link/n4162>.

- **Быстродействие:** атомарные умные указатели обладают огромным преимуществом по сравнению с `atomic_*`-функциями. Атомарные версии предназначены для специального использования и могут внутри иметь `std::atomic_flag` в качестве дешевой спин-блокировки (spinlock)<sup>1</sup>. Делать неатомарные версии указателя потокобезопасными будет слишком дорого, если они используются в однопотоковом сценарии. Это будет иметь негативное влияние на быстродействие.

Аргумент правильности является, пожалуй, наиболее важным. Почему? Ответ лежит в самом предложении N1462. Предложение описывает потокобезопасный односвязный список, поддерживающий вставку, удаление и поиск элементов. Этот односвязный список реализован без блокировки (lock-free).

### 6.2.2.1 Потокобезопасный односвязный список

```
template<typename T> class concurrent_stack {
    struct Node { T t; shared_ptr<Node> next; };
    atomic_shared_ptr<Node> head;
    // in C++11: remove "atomic_" and remember to use the special
    // functions every time you touch the variable
    concurrent_stack( concurrent_stack &) =delete;
    void operator=(concurrent_stack&) =delete;

public:
    concurrent_stack() =default;
    ~concurrent_stack() =default;
    class reference {
        shared_ptr<Node> p;
    public:
        reference(shared_ptr<Node> p_) : p{p_} { }
        T& operator* () { return p->t; }
        T* operator->() { return &p->t; }
    };

    auto find( T t ) const {
        auto p = head.load(); // in C++11: atomic_load(&head)
        while( p && p->t != t )
            p = p->next;
        return reference(move(p));
    }
    auto front() const {
        return reference(head); // in C++11: atomic_load(&head)
    }
    void push_front( T t ) {
        auto p = make_shared<Node>();
        p->t = t;
        p->next = head; // in C++11: atomic_load(&head)
    }
};
```

Потокобезопасный односвязный список

<sup>1</sup> <https://en.wikipedia.org/wiki/Spinlock>.

```

while( !head.compare_exchange_weak(p->next, p) ){ }
// in C++11: atomic_compare_exchange_weak(&head, &p->next, p);
}
void pop_front() {
    auto p = head.load();
    while( p && !head.compare_exchange_weak(p, p->next) ){ }
    // in C++11: atomic_compare_exchange_weak(&head, &p, p->next);
}
};

```

Все изменения, которые требуются для компиляции данного примера под C++11, выделены красным. Реализация с атомарными указателями гораздо проще и поэтому менее подвержена ошибкам. Система типов в C++20 не разрешает использование неатомарных операций над атомарными умными указателями.

Предложение N4162<sup>1</sup> предлагает ввести новые типы `std::atomic_shared_ptr` и `std::atomic_weak_ptr` в качестве атомарных умных указателей. При добавлении их в стандарт ISO C++ они становятся частичными специализациями `std::atomic`, а именно `std::atomic<std::shared_ptr<T>>` и `std::atomic<std::weak_ptr<T>>`.

Следующая программа показывает пять потоков, изменяющих `std::atomic<std::shared_ptr<std::string>>` без синхронизации.

```

1 // atomicSharedPtr.cpp
2
3 #include <iostream>
4 #include <memory>
5 #include <atomic>
6 #include <string>
7 #include <thread>
8
9 int main() {
10
11     std::cout << '\n';
12
13     std::atomic<std::shared_ptr<std::string>> sharString(
14         std::make_shared<std::string>("Zero"));
15
16     std::thread t1([&sharString]{
17         sharString.store(std::make_shared<std::string>(*sharString.load() + "One"));
18     });
19     std::thread t2([&sharString]{
20         sharString.store(std::make_shared<std::string>(*sharString.load() + "Two"));
21     });

```

<sup>1</sup> <http://wg21.link/n4162>.

```

...
22  std::thread t3(&sharString){
23      sharString.store(std::make_shared<std::string>(*sharString.load()+"Three"));
24  };
25  std::thread t4(&sharString){
26      sharString.store(std::make_shared<std::string>(*sharString.load()+"Four"));
27  };
28  std::thread t5(&sharString){
29      sharString.store(std::make_shared<std::string>(*sharString.load()+"Five"));
30  };
31
32  t1.join();
33  t2.join();
34  t3.join();
35  t4.join();
36  t5.join();
37
38  std::cout << *sharString.load() << '\n';
39
40 }

```

Атомарный `std::shared_ptr shaString` (строка 13) инициализируется строкой «Zero». Каждый из пяти потоков от `t1` до `t5` (строки 16–28) добавляет к `sharString` строку, показанную в строке 38. Использование `std::shared_ptr` вместо `std::atomic<std::shared_ptr>` привело бы к гонке данных (data race).

Выполнение этой программы показывает взаимодействие потоков.

```

1 // atomicSharedPtr.cpp
2
3 #include <iostream>
4 #include <memory>
5 #include <atomic>
6 #include <string>
7 #include <thread>
8
9 int main() {
10
11     std::cout << '\n';
12
13     std::atomic<std::shared_ptr<std::string>> sharString(
14         std::make_shared<std::string>("Zero"));
15
16     std::thread t1([&sharString]){
17         sharString.store(std::make_shared<std::string>(*sharString.load()+"One"));
18     };
19     std::thread t2([&sharString]{
20         sharString.store(std::make_shared<std::string>(*sharString.load()+"Two"));
21     });
22     std::thread t3([&sharString]{
23         sharString.store(std::make_shared<std::string>(*sharString.load()+"Three"));
24     });
25     std::thread t4([&sharString]{
26         sharString.store(std::make_shared<std::string>(*sharString.load()+"Four"));
27     });
28     std::thread t5([&sharString]{
29         sharString.store(std::make_shared<std::string>(*sharString.load()+"Five"));
30     });
31
32     #4 t1.join();

```

#### Потокобезопасное изменение `std::string`

Соответственно, атомарные операции для `std::shared_ptr` были объявлены устаревшими в стандарте C++20.

## 6.2.3 Расширения `std::atomic_flag`

Прежде чем я опишу расширения `std::atomic_flag` в C++20, я хотел бы напомнить о `std::atomic_flag` в C++11. Если вы хотите ознакомиться с деталями, то прочтите пост про `std::atomic_flag`<sup>1</sup> в C++11.

### 6.2.3.1 C++11

Класс `std::atomic_flag` – это своего рода атомарное логическое значение. У него есть методы `clear` и `set`. Я буду для простоты называть `false` чистым (`clear`) состоянием и `true` установленным (`set`) состоянием. Метод `clear` позволяет установить его значение в `false`. При помощи метода `test_and_set` вы можете установить значение в `true` и вернуть предыдущее значение. `ATOMIC_FLAG_INIT` позволяет установить `std::atomic` в `false`.

У `std::atomic_flag` есть два полезных свойства:

- это единственное атомарное значение без блокировки;
- это строительный блок для построения более высокоуровневых абстракций.

С вводом стандарта C++11 у класса нет метода, чтобы узнать текущее значение `std::atomic_flag`, не меняя его при этом. Это было исправлено в C++20.

### 6.2.3.2 Расширения C++20

Следующая таблица показывает более продвинутый интерфейс `std::atomic_flag` в C++20.

| Метод                                       | Описание                                                             |
|---------------------------------------------|----------------------------------------------------------------------|
| <code>atomicFlag.clear()</code>             | Очистить атомарный флаг                                              |
| <code>atomicFlag.test_and_set()</code>      | Устанавливает значение атомарного флага и возвращает старое значение |
| <code>atomicFlag.test()(C++20)</code>       | Возвращает значение флага                                            |
| <code>atomicFlag.notify_one()(C++20)</code> | Оповещает один поток, ожидающий этот флаг                            |
| <code>atomicFlag.notify_all(C++20)</code>   | Оповещает все потоки, ожидающие этот флаг                            |
| <code>atomicFlag.wait(bo)(C++20)</code>     | Блокирует поток до оповещения и изменения значения флага             |

Вызов `atomicFlag.test()` возвращает значение атомарного флага, не изменяя его при этом. Кроме того, вы можете использовать атомарный флаг для синхронизации потоков: `atomicFlag.wait()`, `atomicFlag.notify_one()` и `atomicFlag.notify_all()`. Методы `notify_one` и `notify_all` оповещают один или несколько атомарных флагов. Для вызова `atomicFlag.wait(bo)` требуется логическое значение `bo`. Вызов `atomicFlag.wait(bo)` блокирует поток до следующего оповещения или ложного пробуждения (*spurious wakeup*). Далее проверяет, равно ли значение атомарного флага `bo`, и разблокирует поток, если нет. Значение `bo` служит в качестве предиката для защиты от ложных пробуждений. Ложное пробуждение – это ошибочное оповещение.

<sup>1</sup> <https://www.modernescpp.com/index.php/the-atomic-flag>.

По сравнению с тем, как это было реализовано в C++11, конструктор по умолчанию инициализирует флаг значением `false`.

Оставшиеся более мощные атомики могут предоставлять свой функционал за счет использования мьютекса согласно стандарту C++20. В соответствии со стандартом у них есть метод `is_lock_free` для проверки, использует ли этот атомик мьютекс внутри себя. На популярных платформах я всегда получал неправильный ответ. Но вам следует иметь это в виду. Благодаря `constexpr`-функции `atomic<type>::is_always_lock_free` вы можете проверить для каждого атомарного типа, является ли он неблокирующим (lock-free) на всех аппаратных платформах, которые вы используете. Эта проверка возвращает `true`, только если это `true` для всех поддерживаемых аппаратных платформ. Проверка происходит во время компиляции и доступна начиная с C++17.

### 6.2.3.3 Одноразовая синхронизация потоков

Архитектура отправитель–получатель (sender-receiver) довольно распространена для потоков. В подобной архитектуре получатель ожидает уведомления от отправителя перед продолжением работы. Существуют разные способы реализации подобной архитектуры. В C++11 вы можете использовать условные переменные (condition variables) или пары объект-обещание / объект future; в C++20 вы можете использовать `std::atomic_flag`. У каждого из этих подходов есть свои преимущества и недостатки. Поэтому я хочу их сравнить. Я полагаю, что вы не знаете детали использования условных переменных или объектов-обещаний и объектов future. Поэтому дам небольшой обзор.

#### 6.2.3.3.1 Условные переменные

Условная переменная может выполнять роль отправителя или получателя. В качестве отправителя она может оповещать одного или нескольких получателей.

Синхронизация потоков при помощи условных переменных

---

```

1  // threadSynchronizationConditionVariable.cpp
2
3  #include <iostream>
4  #include <condition_variable>
5  #include <mutex>
6  #include <thread>
7  #include <vector>
8
9  std::mutex mut;
10 std::condition_variable condVar;
11
12 std::vector<int> myVec{};
13
```

```
14 void prepareWork() {
15
16     {
17         std::lock_guard<std::mutex> lck(mut);
18         myVec.insert(myVec.end(), {0, 1, 0, 3});
19     }
20     std::cout << "Sender: Data prepared." << '\n';
21     condVar.notify_one();
22 }
23
24 void completeWork() {
25
26     std::cout << "Waiter: Waiting for data." << '\n';
27     std::unique_lock<std::mutex> lck(mut);
28     condVar.wait(lck, []{ return not myVec.empty(); });
29     myVec[2] = 2;
30     std::cout << "Waiter: Complete the work." << '\n';
31     for (auto i: myVec) std::cout << i << " ";
32     std::cout << '\n';
33
34 }
35
36 int main() {
37
38     std::cout << '\n';
39
40     std::thread t1(prepareWork);
41     std::thread t2(completeWork);
42
43     t1.join();
44     t2.join();
45
46     std::cout << '\n';
47
48 }
```

---

У программы есть два дочерних потока `t1` и `t2`. Они получают свое задание `prepareWork` и `completeWork` в строках 40 и 41. Функция `prepareWork` (строка 14) оповещает о том, что она завершила свою работу по подготовке задания: `condVar.notify_one()`. Удерживая блокировку, поток `t2` ожидает оповещения:



`condVar.wait(lck, [{ return not myVec.empty(); }])`. Ожидающий поток всегда выполняет одни и те же шаги. После пробуждения она проверяет предикат, удерживая при этом блокировку (`[{ return not myVec.empty();}]`). Если условие предиката не выполнено, то она снова «засыпает». При выполнении предиката она продолжает свою работу. В данном примере отправляющий поток помещает начальные значения в `std::vector` (строка 18), а принимающий поток завершает их (строка 29).

```

rainer@seminar:~> threadSynchronizationConditionVariables

Waiter: Waiting for data.
Sender: Data prepared.
Waiter: Complete the work.
0 1 2 3

rainer@seminar:~> threadSynchronizationConditionVariables

Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

rainer@seminar:~> threadSynchronizationConditionVariables

Waiter: Waiting for data.
Sender: Data prepared.
Waiter: Complete the work.
0 1 2 3

rainer@seminar:~> threadSynchronizationConditionVariables

Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

rainer@seminar:~> █

```

Синхронизация потоков при помощи условных переменных

У условных переменных есть много врожденных проблем. Например, принимающая сторона может быть «разбужена» без оповещения или же может

потерять оповещение. Первый случай называется ложным пробуждением (spurious wakeup), второй – потерянным пробуждением (lost wakeup). Предикат защищает от них обоих. Оповещение может быть потеряно, когда отправитель посылает оповещение, прежде чем отправитель заходит в состояние ожидания и не использует при этом предикат. Соответственно, получатель ожидает чего-то, что никогда не приходит. Это называется тупиком (deadlock). При изучении вывода программы вы могли заметить, что каждый второй запуск вызвал бы тупик, если бы я не использовал предикат. Хотя, конечно, можно использовать условные переменные и без предикатов.

Если вы хотите узнать про детали архитектуры отправитель–получатель и о подводных камнях использования условных переменных, то прочитайте мою статью «C++ Core Guidelines: Be Aware of the Traps of Condition Variables»<sup>1</sup>.

Давайте теперь реализуем эту же архитектуру при помощи пары объект-обещание / объект future.

#### 6.2.3.3.2 Объекты-обещания и объекты future

Объект-обещание может послать значение, исключение или уведомление связанному с собой объекту future. Ниже приводится пример архитектуры, использующий объекты-обещания и объект future.

Синхронизация потоков при помощи пары объектов-обещание / объектов future

---

```

1  // threadSynchronizationPromiseFuture.cpp
2
3  #include <iostream>
4  #include <future>
5  #include <thread>
6  #include <vector>
7
8  std::vector<int> myVec{};
9
10 void prepareWork(std::promise<void> prom) {
11
12     myVec.insert(myVec.end(), {0, 1, 0, 3});
13     std::cout << "Sender: Data prepared." << '\n';
14     prom.set_value();
15
16 }
17
18 void completeWork(std::future<void> fut){
19
```

---

<sup>1</sup> <https://www.modernescpp.com/index.php/c-core-guidelines-be-aware-of-the-traps-of-condition-variables>.

---

```

20     std::cout << "Waiter: Waiting for data." << '\n';
21     fut.wait();
22     myVec[2] = 2;
23     std::cout << "Waiter: Complete the work." << '\n';
24     for (auto i: myVec) std::cout << i << " ";
25     std::cout << '\n';
26
27 }
28
29 int main() {
30
31     std::cout << '\n';
32
33     std::promise<void> sendNotification;
34     auto waitForNotification = sendNotification.get_future();
35
36     std::thread t1(prepareWork, std::move(sendNotification));
37     std::thread t2(completeWork, std::move(waitForNotification));
38
39     t1.join();
40     t2.join();
41
42     std::cout << '\n';
43
44 }
```

---

Когда вы познакомитесь с этой архитектурой, то узнаете, что синхронизация, по сути, сведена к двум шагам: `prom.set_value()` (строка 14) и `fut.wait()` (строка 21). Я не буду приводить скриншот с результатами выполнения этой программы, поскольку он практически совпадает с результатами предыдущей программы с условными переменными.

Дополнительную информацию по объектам-обещаниям и объектам `future` вы можете найти в описании задач (task)<sup>1</sup>.

#### 6.2.3.3.3 `std::atomic_flag`

Теперь я непосредственно перехожу к тем нововведениям, которые появились в C++20.

---

<sup>1</sup> <https://www.modernescpp.com/index.php/tag/tasks>.

Синхронизация потоков при помощи `std::atomic_flag`

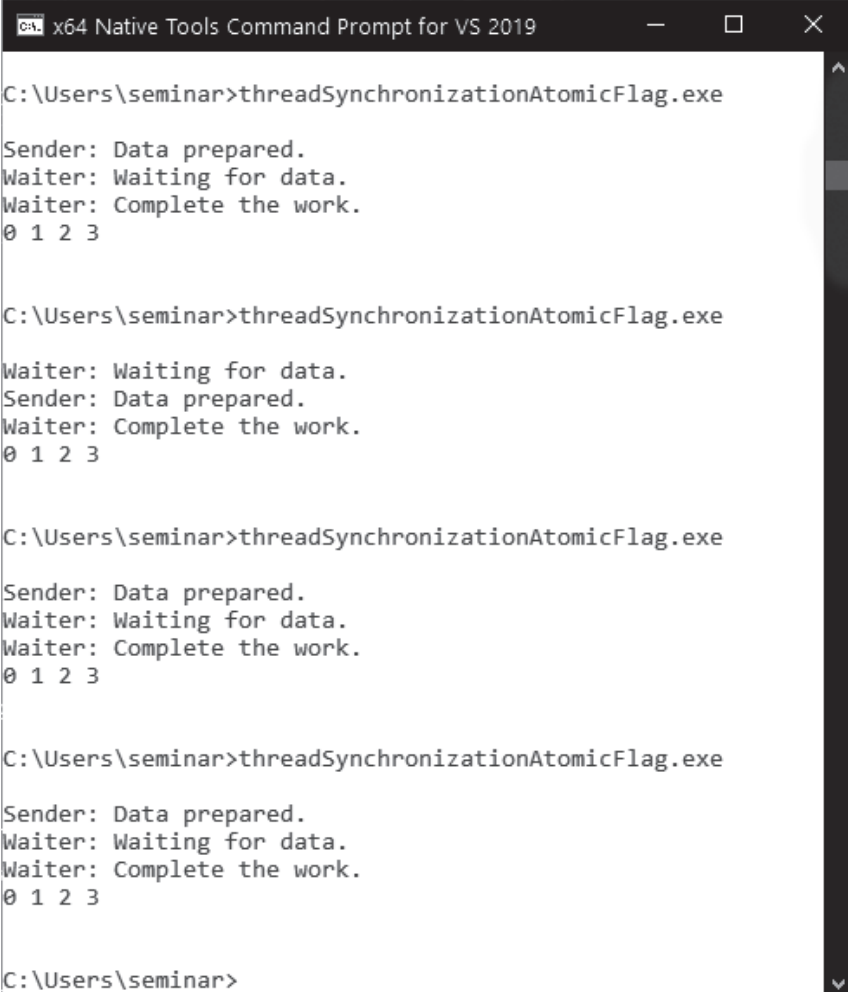
---

```
1  // threadSynchronizationAtomicFlag.cpp
2
3  #include <atomic>
4  #include <iostream>
5  #include <thread>
6  #include <vector>
7
8  std::vector<int> myVec{};
9
10 std::atomic_flag atomicFlag{};
11
12 void prepareWork() {
13
14     myVec.insert(myVec.end(), {0, 1, 0, 3});
15     std::cout << "Sender: Data prepared." << '\n';
16     atomicFlag.test_and_set();
17     atomicFlag.notify_one();
18
19 }
20
21 void completeWork() {
22
23     std::cout << "Waiter: Waiting for data." << '\n';
24     atomicFlag.wait(false);
25     myVec[2] = 2;
26     std::cout << "Waiter: Complete the work." << '\n';
27     for (auto i: myVec) std::cout << i << " ";
28     std::cout << '\n';
29
30 }
31
32 int main() {
33
34     std::cout << '\n';
35
36     std::thread t1(prepareWork);
37     std::thread t2(completeWork);
38
```

```
39     t1.join();
40     t2.join();
41
42     std::cout << '\n';
43
44 }
```

Поток, подготавливающий работу (строка 6), устанавливает `atomicFlag` в `true` и посылает уведомление. Поток, завершающий работу, ожидает уведомления. Он разблокируется, только если `atomicFlag` равен `true`.

Вот несколько запусков программы, собранной при помощи компилятора от Microsoft.



```
C:\Users\seminar>threadSynchronizationAtomicFlag.exe

Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronizationAtomicFlag.exe

Waiter: Waiting for data.
Sender: Data prepared.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronizationAtomicFlag.exe

Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronizationAtomicFlag.exe

Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>
```

## 6.2.4 Расширения `std::atomic`

В стандарте C++20 `std::atomic`, как и `std::atomic_ref`<sup>1</sup>, может быть инстанцирован при помощи типов с плавающей точкой, таких как `float`, `double` и `long double`. Кроме того, `std::atomic_flag`, как и `std::atomic`, может применяться для синхронизации потоков при помощи методов `notify_one`, `notify_all` и `wait`. Уведомление и ожидание доступны для всех частичных и полных специализаций `std::atomic` (логические, целочисленные, числа с плавающей точкой и указатели) и `std::atomic_ref`.

Благодаря `atomic<bool>` можно предыдущую программу `threadSynchronizationAtomicFlag.cpp` переписать немного иначе.

Синхронизация потоков при помощи `std::atomic<bool>`

---

```
1  // threadSynchronizationAtomicBool.cpp
2
3  #include <atomic>
4  #include <iostream>
5  #include <thread>
6  #include <vector>
7
8  std::vector<int> myVec{};
9
10 std::atomic<bool> atomicBool{false};
11
12 void prepareWork() {
13
14     myVec.insert(myVec.end(), {0, 1, 0, 3});
15     std::cout << "Sender: Data prepared." << '\n';
16     atomicBool.store(true);
17     atomicBool.notify_one();
18
19 }
20
21 void completeWork() {
22
23     std::cout << "Waiter: Waiting for data." << '\n';
24     atomicBool.wait(false);
25     myVec[2] = 2;
26     std::cout << "Waiter: Complete the work." << '\n';
27     for (auto i: myVec) std::cout << i << " ";
```

---

<sup>1</sup> <https://en.cppreference.com/w/cpp/atomic/atomic>.

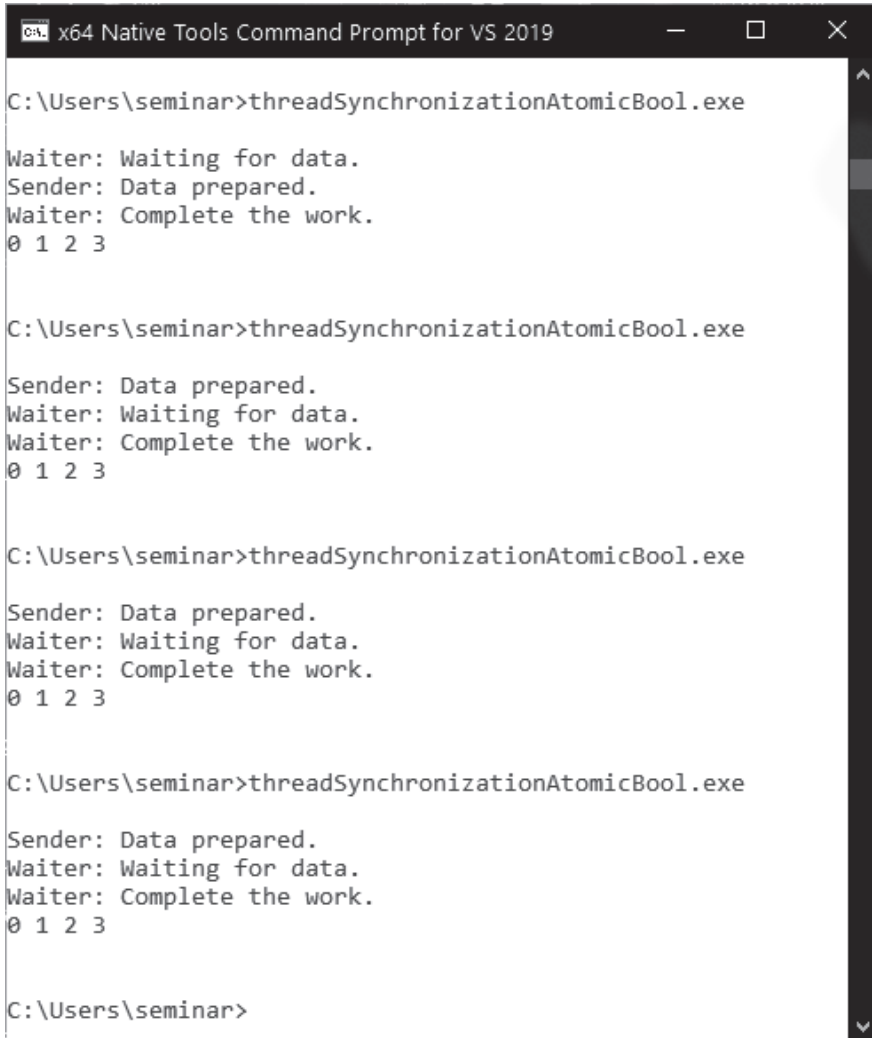
---

```
28     std::cout << '\n';
29
30 }
31
32 int main() {
33
34     std::cout << '\n';
35
36     std::thread t1(prepareWork);
37     std::thread t2(completeWork);
38
39     t1.join();
40     t2.join();
41
42     std::cout << '\n';
43
44 }
```

---

Вызов `atomicBool.wait(false)` блокирует поток, если `atomicBool == false`. Соответственно, вызов `atomicBool.store(true)` (строка 16) устанавливает `atomicBool` в `true` и посылает уведомление.

Результаты нескольких запусков программы, собранных при помощи компилятора от Microsoft, представлены ниже.



```

C:\Users\seminar>threadSynchronizationAtomicBool.exe

Waiter: Waiting for data.
Sender: Data prepared.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronizationAtomicBool.exe

Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronizationAtomicBool.exe

Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronizationAtomicBool.exe

Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>

```

Синхронизация потоков при помощи `std::atomic<bool>`



### Условные переменные vs пара объект-обещание / объект future vs `std::atomic_flag`

Когда вам нужно только однократное уведомление, такое как в предыдущей программе `threadSynchronizationConditionVariable.cpp`, то выбор объект-обещание / объект future будет лучше, чем выбор условных переменных. Объекты-обещания и объекты future не подвержены случайным и потерянным пробуждениям. У них есть только один недостаток: они могут быть использованы лишь один раз.

Я не уверен, буду ли я использовать пару `std::promise/std::future` или атомики, например `std::atomic<bool>` или `std::atomic_flag`, для подобного простого случая синхронизации. Все они гарантируют потокобезопасность по определению и не требуют спе-



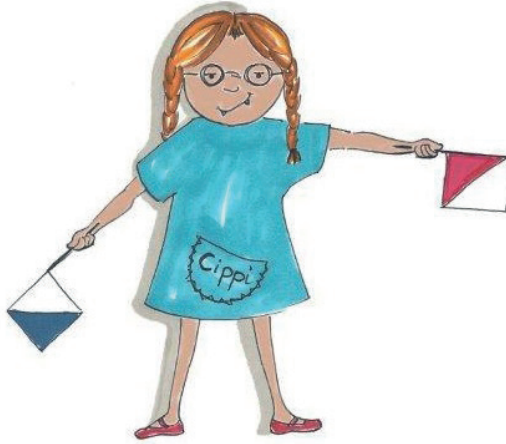
циального механизма для защиты. Проще будет использовать пару `std::promise/std::future`, но атомики будут, скорее всего, быстрее. Я уверен только в том, что не стал бы использовать условную переменную, если без этого можно обойтись.



### Краткая информация

- ♦ `std::atomic_ref` применяет атомарные операции к объекту, на который он ссылается. Гарантируется одновременное чтение и запись объектов, на которые указывает ссылка, без состояния гонки (race condition). Время жизни объекта, на который указывает ссылка, должно превышать время жизни самого `std::atomic_ref`.
- ♦ `std::shared_ptr` состоит из управляющего блока и своего ресурса. Управляющий блок потокобезопасен, но доступ к ресурсу – нет. В C++20 имеются атомарные общие указатели: `std::atomic<std::shared_ptr<T>>` и `std::atomic<std::weak_ptr<T>>`.
- ♦ `std::atomic_flag` – это своего рода атомарное логическое значение, являющееся единственной структурой данных в C++ без блокировок. Его ограниченный интерфейс был расширен в C++20. Вы можете получить его значение и использовать его для синхронизации потоков.
- ♦ `std::atomic`, введенный в стандарте C++11, получил в C++20 многочисленные улучшения. Вы можете специализировать `std::atomic` для значений с плавающей точкой и использовать его для синхронизации потоков.

## 6.3 Семафоры



Сиппи управляет поездом

Семафоры являются механизмом синхронизации, используемым для управления одновременным доступом к общему ресурсу. Считающий семафор – это специальный семафор со счетчиком, большим нуля. Счетчик инициализируется в конструкторе. Этот счетчик уменьшается при вызовах метода `acquire()` и связанных с ним методов и увеличивается при вызовах метода `release()`. Если поток пытается получить семафор, счетчик которого равен нулю, то этот поток блокируется до тех пор, пока другой поток не увеличит счетчик, освобождая семафор.

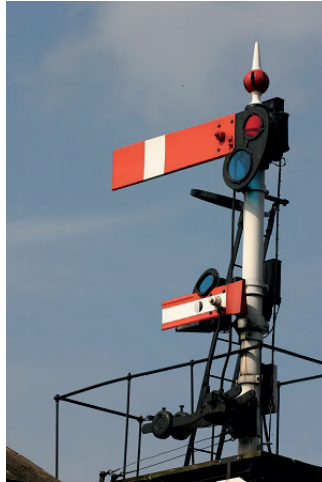


### Едсгер Дейкстра изобрел семафоры

Голландский ученый Эдсгер Вибе Дейкстра<sup>1</sup> в 1965 году представил понятие семафора. Семафор – это структура данных с очередью и счетчиком. Счетчик инициализируется начальным значением, большим или равным нулю. Он поддерживает две операции: `wait` и `signal`. Операция `wait` получает семафор и уменьшает счетчик. Она блокирует поток, если при получении счетчик равен нулю. Операция `signal` освобождает семафор и увеличивает счетчик. Блокированные потоки добавляются в очередь, чтобы избежать голодания (*starvation*)<sup>2</sup>.

<sup>1</sup> [https://en.wikipedia.org/wiki/Edsger\\_W.\\_Dijkstra](https://en.wikipedia.org/wiki/Edsger_W._Dijkstra).

<sup>2</sup> [https://en.wikipedia.org/wiki/Starvation\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Starvation_(computer_science)).



Семафор

Этот рисунок принадлежит AmosWolfe из английской Википедии<sup>1</sup>.

Стандарт C++20 поддерживает `std::binary_semaphore`, являющийся другим именем для `std::counting_semaphore<1>`. В этом случае максимальное значение должно быть не менее 1. `std::counting_semaphore` может использоваться для реализации блокировок (locks)<sup>2</sup>.

```
using binary_semaphore = std::counting_semaphore<1>;
```

В отличие от `std::mutex`, `std::counting_semaphore` не привязан к потоку. Это значит, что получение и освобождение семафора могут происходить в разных потоках. В следующей таблице приводится интерфейс `std::counting_semaphore`.

| Метод                                       | Описание                                                                                                          |
|---------------------------------------------|-------------------------------------------------------------------------------------------------------------------|
| <code>sem.max()(static)</code>              | Возвращает максимальное значение счетчика                                                                         |
| <code>sem.release(upd = 1)</code>           | Увеличивает счетчик на <code>upd</code> и соответственно разблокирует потоки, получающие семафор <code>sem</code> |
| <code>sem.acquire()</code>                  | Уменьшает счетчик на 1 или блокирует до тех пор, пока счетчик не станет больше 0                                  |
| <code>sem.try_acquire()</code>              | Пытается уменьшить счетчик на 1, если он больше 0                                                                 |
| <code>sem.try_acquire_for(relTime)</code>   | Пытается уменьшить счетчик или блокирует не более чем на <code>absTime</code> , если счетчик равен 0              |
| <code>sem.try_acquire_until(absTime)</code> | Пытается уменьшить счетчик или блокирует не более чем <code>absTime</code> , если счетчик равен 0                 |

<sup>1</sup> <https://commons.wikimedia.org/w/index.php?curid=1972304>.

<sup>2</sup> [https://en.cppreference.com/w/cpp/named\\_req/BasicLockable](https://en.cppreference.com/w/cpp/named_req/BasicLockable).

Конструкция `std::counting_semaphore<10> sem(5)` создает семафор `sem` с максимальным счетчиком как минимум 10 и текущим значением счетчика, равным 5. Вызов `sem.max()` возвращает наименьшее максимальное значение. Вызов `sem.try_acquire_for(relTime)` получает на вход промежуток времени<sup>1</sup>; метод `sem.try_acquire_until(absTime)` получает на вход момент времени<sup>2</sup>. Три вызова – `sem.try_acquire`, `sem.try_acquire_for` и `sem.try_acquire_until` – возвращают логическое значение, говорящее об успешности вызова.

Семафоры обычно используются в архитектурах отправитель–получатель. Например, инициализация семафора `sem` нулем блокирует вызов получателя `sem.acquire()` до тех пор, пока отправитель не вызовет `sem.release()`. Соответственно, получатель ждет уведомления от отправителя. Ранее рассмотренная программа с синхронизацией потоков легко может быть реализована при помощи семафора.

Синхронизация потоков при помощи `std::counting_semaphore`

---

```
1  // threadSynchronizationSemaphore.cpp
2
3  #include <iostream>
4  #include <semaphore>
5  #include <thread>
6  #include <vector>
7
8  std::vector<int> myVec{};
9
10 std::counting_semaphore<1> prepareSignal(0);
11
12 void prepareWork() {
13
14     myVec.insert(myVec.end(), {0, 1, 0, 3});
15     std::cout << "Sender: Data prepared." << '\n';
16     prepareSignal.release();
17 }
18
19 void completeWork() {
20
21     std::cout << "Waiter: Waiting for data." << '\n';
22     prepareSignal.acquire();
23     myVec[2] = 2;
24     std::cout << "Waiter: Complete the work." << '\n';
```

---

<sup>1</sup> <https://en.cppreference.com/w/cpp/chrono/duration>.

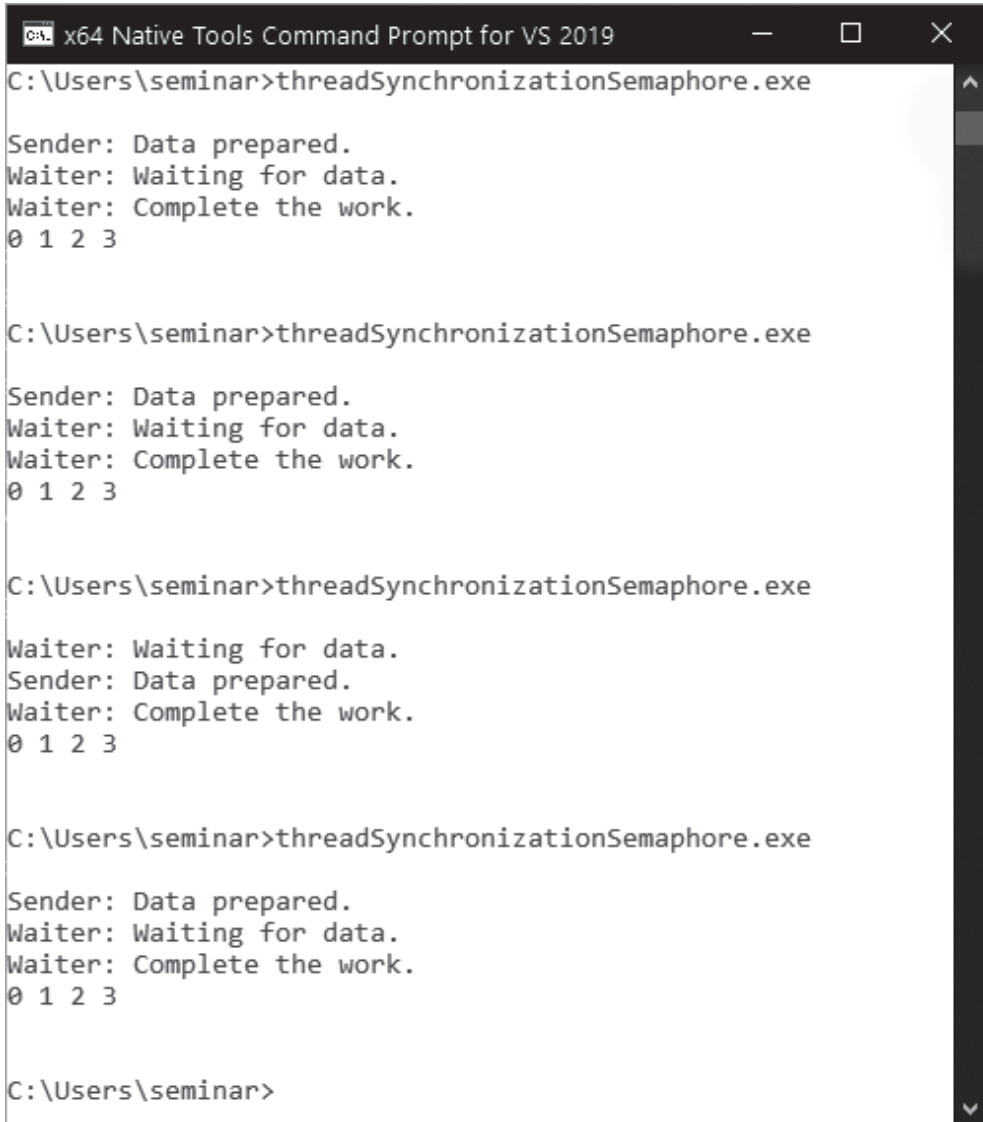
<sup>2</sup> [https://en.cppreference.com/w/cpp/chrono/time\\_point](https://en.cppreference.com/w/cpp/chrono/time_point).

---

```
25     for (auto i: myVec) std::cout << i << " ";
26     std::cout << '\n';
27
28 }
29
30 int main() {
31
32     std::cout << '\n';
33
34     std::thread t1(prepareWork);
35     std::thread t2(completeWork);
36
37     t1.join();
38     t2.join();
39
40     std::cout << '\n';
41
42 }
```

---

Объявление `std::counting_semaphore prepareSignal` (строка 10) создает семафор, счетчик которого может принимать значения 0 и 1. Это значит, что вызов `prepareSignal.release()` устанавливает счетчик в 1 (строка 16) и разблокирует вызов `prepareSignal.acquire()` (строка 22).



```
x64 Native Tools Command Prompt for VS 2019
C:\Users\seminar>threadSynchronizationSemaphore.exe

Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronizationSemaphore.exe

Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronizationSemaphore.exe

Waiter: Waiting for data.
Sender: Data prepared.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronizationSemaphore.exe

Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>
```

Синхронизация потоков при помощи семафора



### Краткая информация

- ♦ Семафоры – это механизм синхронизации, используемый для управления одновременным доступом к общему ресурсу.
- ♦ Считающий семафор в C++20 содержит счетчик. Получение семафора уменьшает этот счетчик на 1, освобождение увеличивает его на единицу. Если поток пытается получить семафор, счетчик которого равен 0, то данный поток блокируется до тех пор, пока другой поток не увеличит счетчик, освобождая семафор.

## 6.4 Защелки и барьеры



Сиппи ждет у шлагбаума

Защелки (latches) и барьеры (barriers) – это координационные типы, позволяющие отдельным потокам блокироваться до тех пор, пока счетчик не станет нулем. Стандартом C++20 предусмотрено два варианта защелок и барьеров: `std::latch` и `std::barrier`. Одновременные вызовы методов `std::latch` и `std::barrier` не приводят к состоянию гонки (data race).

Для начала разберем два вопроса:

1. В чем разница между этими двумя механизмами для координации потоков? Вы можете использовать `std::latch` только один раз, но `std::barrier` можно использовать многократно. `std::latch` – это удобный механизм для управления одной задачей с помощью нескольких потоков. `std::barrier` помогает управлять повторяющимися задачами с помощью нескольких потоков. Кроме того, `std::barrier` позволяет выполнить функцию в так называемом шаге завершения. Шаг завершения – это состояние, когда счетчик становится равным нулю.
2. Что такого делают защелки и барьеры, что не может быть сделано в C++11 и C++14 при помощи объектов `future`, потоков или условных переменных вместе с блокировками (locks)? Защелки и барьеры не имеют каких-то новых случаев для использования, они просто гораздо легче в применении. Кроме того, они предлагают большее быстродействие, поскольку часто используют внутри себя неблокирующие механизмы.

### 6.4.1 `std::latch`

Теперь давайте рассмотрим интерфейс `std::latch`.

Методы объекта `std::latch` `lat`

| Метод                                     | Описание                                                                                                                                                         |
|-------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>lat.count_down(upd = 1)</code>      | Атомарно уменьшает счетчик на <code>upd</code> без блокировки вызывающего потока                                                                                 |
| <code>lat.try_wait()</code>               | Возвращает <code>true</code> , если <code>counter == 0</code>                                                                                                    |
| <code>lat.wait()</code>                   | Немедленно возвращает управление, если <code>counter == 0</code> . Если нет, то блокируется до того момента, пока <code>counter</code> не будет <code>= 0</code> |
| <code>lat.arrive_and_wait(upd = 1)</code> | Эквивалентно <code>count_down(upd); wait();</code>                                                                                                               |

Значением по умолчанию для `upd` является 1. Когда `upd` больше, чем значение счетчика, или отрицательно, то поведение программы не определено. Вызов `lat.try_wait()` никогда не ждет (как на это намекает его имя).

Следующая программа `bossWorkers.cpp` использует два `std::latch` для построения архитектуры босс–исполнитель (`boss-worker workflow`). Я синхронизирую вывод в `std::cout` при помощи функции `synchronizedOut` (строка 13). Синхронизация облегчает отслеживание взаимодействия потоков.

Архитектура босс–исполнитель с двумя `std::latch`

---

```
1  // bossWorkers.cpp
2
3  #include <iostream>
4  #include <mutex>
5  #include <latch>
6  #include <thread>
7
8  std::latch workDone(6);
9  std::latch goHome(1);
10
11 std::mutex coutMutex;
12
13 void synchronizedOut(const std::string& s) {
14     std::lock_guard<std::mutex> lo(coutMutex);
15     std::cout << s;
16 }
17
18 class Worker {
19 public:
20     Worker(std::string n): name(n) { }
21
```



```

22     void operator() (){
23         // notify the boss when work is done
24         synchronizedOut(name + ": " + "Work done!\n");
25         workDone.count_down();
26
27         // waiting before going home
28         goHome.wait();
29         synchronizedOut(name + ": " + "Good bye!\n");
30     }
31     private:
32         std::string name;
33 };
34
35 int main() {
36
37     std::cout << '\n';
38
39     std::cout << "BOSS: START WORKING! " << '\n';
40
41     Worker herb(" Herb");
42     std::thread herbWork(herb);
43
44     Worker scott(" Scott");
45     std::thread scottWork(scott);
46
47     Worker bjarne(" Bjarne");
48     std::thread bjarneWork(bjarne);
49
50     Worker andrei(" Andrei");
51     std::thread andreiWork(andrei);
52
53     Worker andrew(" Andrew");
54     std::thread andrewWork(andrew);
55
56     Worker david(" David");
57     std::thread davidWork(david);
58
59     workDone.wait();
60

```

```

61     std::cout << '\n';
62
63     goHome.count_down();
64
65     std::cout << "BOSS: GO HOME!" << '\n';
66
67     herbWork.join();
68     scottWork.join();
69     bjarneWork.join();
70     andreiWork.join();
71     andrewWork.join();
72     davidWork.join();
73
74 }

```

Архитектура очень проста. Шесть исполнителей – herb, scott, bjarne, andrei, andrew и david (строки 41–57) – должны выполнить свою работу. Когда каждый из них завершит свою работу, он уменьшает счетчик `std::latch workDone` (строка 25). Босс (главный поток) блокируется в строке 59 до тех пор, пока счетчик не станет равным нулю. Когда счетчик становится равным нулю, босс использует `std::latch goHome`, чтобы сообщить своим исполнителям, что они могут завершаться. В этом случае сначала счетчик равен 1 (строка 9). Вызов `goHome.wait()` блокирует поток до тех пор, пока счетчик не станет равным 0.

```

C:\Users\seminar>bossWorkers.exe

BOSS: START WORKING!
  Herb: Work done!
    Andrei: Work done!
      Bjarne: Work done!
        Andrew: Work done!
          David: Work done!
            Scott: Work done!

BOSS: GO HOME!
  David: Good bye!
    Andrew: Good bye!
      Scott: Good bye!
        Andrei: Good bye!
          Bjarne: Good bye!
            Herb: Good bye!

C:\Users\seminar>

```

Архитектура босс–исполнитель с двумя `std::latch`

Если подумать об этой архитектуре, то вы можете заметить, что вся работа могла быть выполнена без босса. Вот как это можно сделать.

Архитектура исполнителей с `std::latch`

---

```

1  // workers.cpp
2
3  #include <iostream>
4  #include <barrier>
5  #include <mutex>
6  #include <thread>
7
8  std::latch workDone(6);
9  std::mutex coutMutex;
10
11 void synchronizedOut(const std::string& s) {
12     std::lock_guard<std::mutex> lo(coutMutex);
13     std::cout << s;
14 }
15
16 class Worker {
17 public:
18     Worker(std::string n): name(n) { }
19
20     void operator() () {
21         synchronizedOut(name + ": " + "Work done!\n");
22         workDone.arrive_and_wait(); // wait until all work is done
23         synchronizedOut(name + ": " + "See you tomorrow!\n");
24     }
25 private:
26     std::string name;
27 };
28
29 int main() {
30
31     std::cout << '\n';
32
33     Worker herb(" Herb");
34     std::thread herbWork(herb);
35

```

```
36     Worker scott("    Scott");
37     std::thread scottWork(scott);
38
39     Worker bjarne("    Bjarne");
40     std::thread bjarneWork(bjarne);
41
42     Worker andrei("    Andrei");
43     std::thread andreiWork(andrei);
44
45     Worker andrew("    Andrew");
46     std::thread andrewWork(andrew);
47
48     Worker david("    David");
49     std::thread davidWork(david);
50
51     herbWork.join();
52     scottWork.join();
53     bjarneWork.join();
54     andreiWork.join();
55     andrewWork.join();
56     davidWork.join();
57
58 }
```

---

Очень немного можно сказать об этой архитектуре. Вызов `wordDone.arrive_and_wait()` (строка 22) эквивалентен `count_down(upd); wait();`. Это значит, что исполнители сами себя координируют и босс уже не нужен, в отличие от предыдущего примера `bossWorkers.cpp`.

```

C:\Users\seminar>workers.exe

Herb: Work done!
  Andrei: Work done!
    Scott: Work done!
      Andrew: Work done!
        David: Work done!
          Bjarne: Work done!
            Bjarne: See you tomorrow!
              Andrew: See you tomorrow!
                Andrei: See you tomorrow!
                  David: See you tomorrow!
                    Scott: See you tomorrow!
                      Herb: See you tomorrow!

C:\Users\seminar>

```

Архитектура исполнителей с `std::latch`

Класс `std::barrier` очень похож на `std::latch`.

## 6.4.2 `std::barrier`

Есть два отличия между `std::barrier` и `std::latch`. Во-первых, вы можете использовать `std::barrier` более одного раза, и, во-вторых, вы можете установить счетчик для следующего шага (итерации). После того как счетчик становится равным 0, сразу начинается шаг завершения (completion step). На этом шаге вызывается вызываемый объект. Класс `std::barrier` получает этот вызываемый объект в своем конструкторе.

Шаг завершения выполняет следующие действия:

1. Все потоки блокируются.
2. Произвольный поток разблокируется и выполняет вызываемый объект.
3. Если шаг завершения выполнен, то все потоки разблокируются.

Методы объекта `std::barrier bar`

| Метод                              | Описание                                                   |
|------------------------------------|------------------------------------------------------------|
| <code>bar.arrive(upd)</code>       | Атомарно уменьшить счетчик на <code>upd</code>             |
| <code>bar.wait()</code>            | Заблокировать до окончания шага завершения                 |
| <code>bar.arrive_and_wait()</code> | Эквивалентно <code>wait(arrive())</code>                   |
| <code>bar.arrive_and_drop()</code> | Уменьшает счетчик для текущей и последующей фаз на единицу |
| <code>std::barrier::max</code>     | Максимальное значение, поддерживаемое реализацией барьера  |

Вызов `bar.arrive_and_drop()` значит, что счетчик уменьшается на единицу для следующей фазы. Программа `fullTimePartTimeWorkers.cpp` уменьшает количество исполнителей на второй фазе.

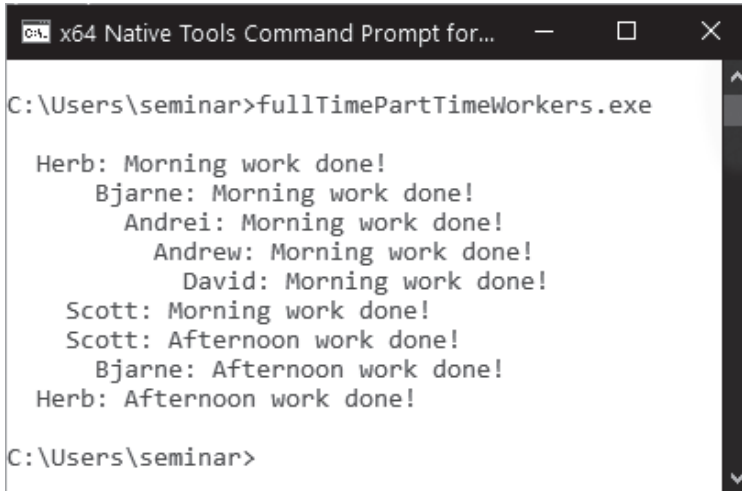
Исполнители на полную ставку и на частичную

---

```
1  // fullTimePartTimeWorkers.cpp
2
3  #include <iostream>
4  #include <barrier>
5  #include <mutex>
6  #include <string>
7  #include <thread>
8
9  std::barrier workDone(6);
10 std::mutex coutMutex;
11
12 void synchronizedOut(const std::string& s) {
13     std::lock_guard<std::mutex> lo(coutMutex);
14     std::cout << s;
15 }
16
17 class FullTimeWorker {
18 public:
19     FullTimeWorker(std::string n): name(n) { }
20
21     void operator() () {
22         synchronizedOut(name + ": " + "Morning work done!\n");
23         workDone.arrive_and_wait(); // Wait until morning work is done
24         synchronizedOut(name + ": " + "Afternoon work done!\n");
25         workDone.arrive_and_wait(); // Wait until afternoon work is done
26     }
27 private:
28     std::string name;
29 };
30
31
32 class PartTimeWorker {
33 public:
34     PartTimeWorker(std::string n): name(n) { }
35
```

```
36     void operator() () {
37         synchronizedOut(name + ": " + "Morning work done!\n");
38         workDone.arrive_and_drop(); // Wait until morning work is done
39     }
40 private:
41     std::string name;
42 };
43
44 int main() {
45
46     std::cout << '\n';
47
48     FullTimeWorker herb("  Herb");
49     std::thread herbWork(herb);
50
51     FullTimeWorker scott("  Scott");
52     std::thread scottWork(scott);
53
54     FullTimeWorker bjarne("    Bjarne");
55     std::thread bjarneWork(bjarne);
56
57     PartTimeWorker andrei("        Andrei");
58     std::thread andreiWork(andre);
59
60     PartTimeWorker andrew("        Andrew");
61     std::thread andrewWork(andrew);
62
63     PartTimeWorker david("        David");
64     std::thread davidWork(david);
65
66     herbWork.join();
67     scottWork.join();
68     bjarneWork.join();
69     andreiWork.join();
70     andrewWork.join();
71     davidWork.join();
72
73 }
```

Эта архитектура состоит из двух типов исполнителей: на полный рабочий день (строка 17) и с частичной занятостью (строка 32). Исполнители с частичной занятостью работают только утром, исполнители с полной занятостью – утром и днем. Исполнители с полным рабочим днем вызывают `workDone.arrive_and_wait()` (строки 23 и 25) два раза. Исполнители с частичной занятостью вызывают `workDone.arrive_and_drop()` (строка 38) только один раз. Вызов `workDone.arrive_and_drop()` приводит к тому, что исполнители на неполную ставку пропускают работу днем. Соответственно, счетчик в начале (утром) равен 6, а на следующей фазе (днем) он равен 3.



```

C:\Users\seminar>fullTimePartTimeWorkers.exe

Herb: Morning work done!
  Bjarne: Morning work done!
    Andrei: Morning work done!
      Andrew: Morning work done!
        David: Morning work done!
          Scott: Morning work done!
            Scott: Afternoon work done!
              Bjarne: Afternoon work done!
                Herb: Afternoon work done!

C:\Users\seminar>
  
```

Исполнители на полную ставку и на частичную

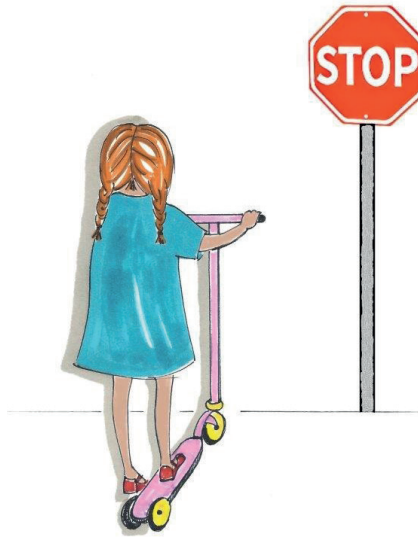


### Краткая информация

- ♦ Защелки и барьеры являются специальными типами данных, применяемыми для координации потоков. Они позволяют блокировать некоторые потоки до тех пор, пока счетчик не станет равным 0. Вы можете использовать `std::latch` всего один раз, а `std::barrier` много раз.
- ♦ `std::latch` удобен для управления одной задачей несколькими потоками, а `std::barrier` управляет повторяющимися задачами несколькими потоками.



## 6.5 Координированное прерывание



Сиппи останавливается перед знаком STOP

Дополнительная функциональность по координированному прерыванию (cooperative interruption) потоков основывается на классах `std::stop_source`, `std::stop_token` и `std::stop_callback`. Классы `std::thread`, `std::condition_variable_any` поддерживают координированное прерывание.

Для начала я отвечу на вопрос, почему уничтожение потока – это не самая хорошая идея?



### Уничтожение потока опасно

Уничтожение потока опасно потому, что вы не знаете его текущее состояние. Есть два возможных нежелательных результата:

- ♦ поток только наполовину выполнил свою работу. Вы не знаете состояние его работы и, соответственно, состояние своей программы. В результате программа работает неопределенно;
- ♦ этот поток может быть в критической секции и может заблокировать мьютекс. Уничтожение потока в то время, когда он заблокировал мьютекс, с большой вероятностью приведет к ситуации тупика (deadlock).

Классы `std::stop_source`, `std::stop_token` и `std::stop_callback` позволяют потоку асинхронно запросить остановку его выполнения или узнать, был ли получен сигнал на остановку. Экземпляр класса `std::stop_token` может быть передан операции и использован далее для получения токена для запроса на прерывание или для регистрации процедуры обратного вызова (callback) через `std::stop_callback`. Запрос на прерывание посылается через `std::stop_source`. Он действует все ассоциированные `std::stop_token`. Три класса – `std::stop_source`,

`std::stop_token` и `std::stop_callback` – разделяют владение ассоциированным состоянием для прерывания.

В следующих подразделах я разберу координированное прерывание более подробно.

### 6.5.1 `std::stop_source`

Вы можете создать `std::stop_source` двумя способами.

Конструкторы `std::stop_source`

---

```
1  std::stop_source();
2  explicit std::stop_source(std::nostopstate_t) noexcept;
```

---

Конструктор по умолчанию (строка 1) строит `std::stop_source` с новым состоянием прерывания (stop state). Конструктор, принимающий на вход `std::nostopstate_t` (строка 2), строит пустой `std::stop_source` без ассоциированного состояния остановки.

Компонент `std::stop_source src` предоставляет следующие методы для обработки запросов на прерывание.

Методы объекта `std::stop_source src`

| Метод                             | Описание                                                                                                                                                                                                                |
|-----------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>src.get_token()</code>      | Если <code>stop_possible()</code> , то возвращает <code>stop_token</code> для ассоциированного состояния остановки. В противном случае возвращает пустой <code>stop_token</code> (созданный конструктором по умолчанию) |
| <code>src.stop_possible()</code>  | <code>true</code> , если у <code>src</code> можно потребовать прерывание                                                                                                                                                |
| <code>src.stop_requested()</code> | <code>true</code> , если <code>stop_possible()</code> и <code>request_stop()</code> были вызваны одним из владельцев                                                                                                    |
| <code>src.request_stop()</code>   | Вызывает запрос на прерывание, если <code>src.stop_possible()</code> и <code>!src.stop_requested()</code> . В противном случае ничего не делает                                                                         |

Вызов `src.get_token()` возвращает стоп-токен `token`. Благодаря ему можно проверить, был ли сделан запрос на прерывание или мог бы он быть сделан при помощи связанного стоп-источника `src`. Стоп-токен `token` наблюдает за стоп-источником `src`.

Вызов `src.request_stop()` виден для всех `std::stop_source` и `std::stop_token` того же стоп-состояния. Также любая зарегистрированная функция обратного вызова (callback), связанная с `std::stop_token`, или любая `std::conditional_variable_any`, ожидающая соответствующего `std::stop_token`, будет активирована. Когда сделан запрос на прерывание (stop request), то его нельзя отозвать назад. Вызовы, такие как `std::stop_requested()` или `std::stop_possible()`, атомарны.

Вызов `std::stop_requested()` возвращает `true`, если у `src` есть ассоциированное стоп-состояние и не было запроса на прерывание ранее. Вызов `src.request_stop()` успешен и возвращает `true`, если у `src` есть ассоциированное стоп-состояние и не было запроса на прерывание ранее.

## 6.5.2 std::stop\_token

Класс `std::stop_token` – это фактически потокобезопасный «вид» на ассоциированное стоп-состояние. Этот объект обычно можно получить от `std::jthread` или `std::stop_source src` через `src.get_token()`. Подобные объекты разделяют совместное стоп-состояние `std::jthread` или `std::stop_source`.

Благодаря `std::stop_token` вы можете проверить, был ли сделан запрос на прерывание.

Также `std::stop_token` может быть передан в конструктор `std::stop_callback` или в ожидающие функции `std::condition_variable_any`.

Методы `std::stop_token` `token`

| Метод                               | Описание                                                                                                                                                     |
|-------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>token.stop_possible()</code>  | Возвращает <code>true</code> , если для <code>token</code> есть ассоциированное стоп-состояние                                                               |
| <code>token.stop_requested()</code> | Возвращает <code>true</code> , если был вызван <code>request_stop()</code> для ассоциированного <code>std::stop_source src</code> , иначе <code>false</code> |

Вызов `token.stop_possible()` также возвращает `true`, если запрос на прерывание уже был сделан. Созданный при помощи конструктора по умолчанию, `std::stop_token` не имеет ассоциированного стоп-состояния.

Вызов `token.stop_requested()` возвращает `true`, когда токен имеет ассоциированное стоп-состояние и уже получил запрос на прерывание.

Если необходимо временно заблокировать `std::stop_token`, то его можно заменить на токен, созданный при помощи конструктора по умолчанию. Токен, созданный при помощи конструктора по умолчанию, не имеет ассоциированного стоп-состояния. Следующий пример кода показывает, как выключать и включать возможность потока принимать запросы на прерывание.

Временно заблокировать стоп-токен

---

```

1  std::jthread jthr([](std::stop_token token) {
2      ...
3      std::stop_token interruptDisabled;
4      std::swap(token, interruptDisabled);
5      ...
6      std::swap(token, interruptDisabled);
7      ...
8  }
```

---

У `std::stop_token interruptDisabled` нет ассоциированного стоп-состояния. Это значит, что поток `jthr` может принимать запросы на прерывание во всех строках, кроме 4 и 5.

## 6.5.3 std::stop\_callback

Следующий пример кода показывает использование `std::stop_callback`.

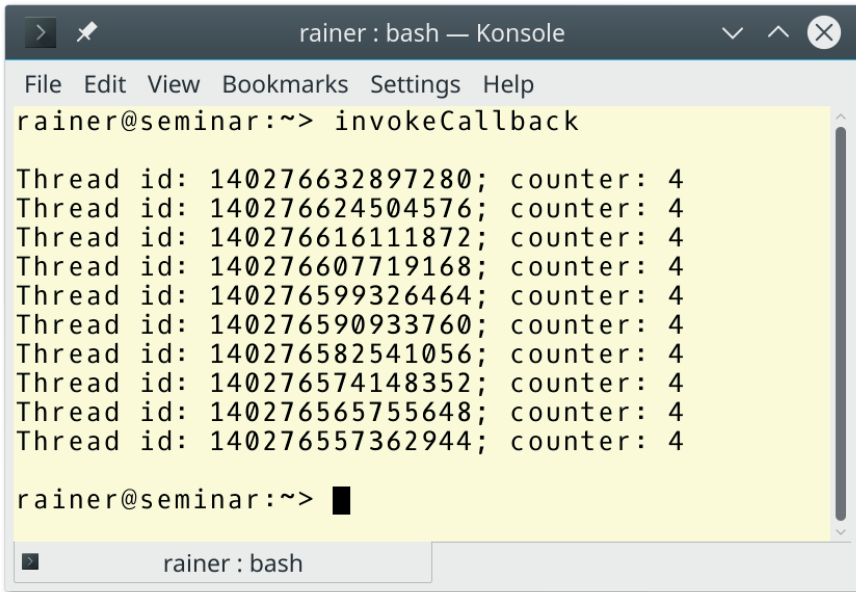
Использование функций обратного вызова

---

```
1  // invokeCallback.cpp
2
3  #include <atomic>
4  #include <chrono>
5  #include <iostream>
6  #include <thread>
7  #include <vector>
8
9  using namespace std::literals;
10
11 auto func = [](std::stop_token token) {
12     std::atomic<int> counter{0};
13     auto thread_id = std::this_thread::get_id();
14     std::stop_callback callBack(token, [&counter, thread_id] {
15         std::cout << "Thread id: " << thread_id
16                 << "; counter: " << counter << '\n';
17     });
18     while (counter < 10) {
19         std::this_thread::sleep_for(0.2s);
20         ++counter;
21     }
22 };
23
24 int main() {
25
26     std::cout << '\n';
27
28     std::vector<std::jthread> vecThreads(10);
29     for(auto& thr: vecThreads) thr = std::jthread(func);
30
31     std::this_thread::sleep_for(1s);
32
33     for(auto& thr: vecThreads) thr.request_stop();
34
35     std::cout << '\n';
36
37 }
```

---

Каждый из 10 потоков вызывает лямбда-функцию `func` (строки 11–22). Функция обратного вызова в строках 14–17 выводит идентификатор потока и счетчик. Благодаря 1-секундной задержке главного потока и задержке других дочерних потоков, когда эти функции вызываются, счетчик равен четырем. Вызов `thr.request_stop()` приводит к вызову функции в каждом потоке.



```

rainer@seminar:~> invokeCallback

Thread id: 140276632897280; counter: 4
Thread id: 140276624504576; counter: 4
Thread id: 140276616111872; counter: 4
Thread id: 140276607719168; counter: 4
Thread id: 140276599326464; counter: 4
Thread id: 140276590933760; counter: 4
Thread id: 140276582541056; counter: 4
Thread id: 140276574148352; counter: 4
Thread id: 140276565755648; counter: 4
Thread id: 140276557362944; counter: 4

rainer@seminar:~>

```

Использование функций обратного вызова

### 6.5.3.1 Присоединение потоков

Класс `std::jthread` – это просто `std::thread` с дополнительной функциональностью для вызова прерывания (`signal an interrupt`) и автоматического `join()`. Для поддержки этой функциональности используется `std::stop_token`.

Методы объекта `std::jthread` `jthr` для работы со стоп-токеном

| Метод                            | Описание                                                                                        |
|----------------------------------|-------------------------------------------------------------------------------------------------|
| <code>t.get_stop_source()</code> | Возвращает объект <code>std::stop_source</code> , ассоциированный с разделяемым стоп-состоянием |
| <code>t.get_stop_token()</code>  | Возвращает объект <code>std::stop_token</code> , ассоциированный с разделяемым стоп-состоянием  |
| <code>t.request_stop()</code>    | Запрашивает выполнение остановки через разделяемое стоп-состояние                               |

### 6.5.3.2 Новые перегруженные функции wait для condition\_variable\_any

Класс `std::condition_variable_any` – это обобщение `std::condition_variable`<sup>1</sup>. Класс `std::condition_variable` требует `std::unique_lock<std::mutex>`, но `std::condition_variable_any` может работать с любыми блокировками (`lock`) `lo`, поддерживающими `lo.lock()` и `lo.unlock()`.

Для `std::condition_variable_any` есть три варианта функции ожидания: `wait`, `wait_for` и `wait_until`. Они все принимают `std::stop_token`.

Три новые операции ожидания (`wait`)

---

```
1  template <class Predicate>
2  bool wait(Lock& lock,
3           stop_token stoken,
4           Predicate pred);
5
6  template <class Rep, class Period, class Predicate>
7  bool wait_for(Lock& lock,
8               stop_token stoken,
9               const chrono::duration<Rep, Period>& rel_time,
10              Predicate pred);
11
12 template <class Clock, class Duration, class Predicate>
13 bool wait_until(Lock& lock,
14                stop_token stoken,
15                const chrono::time_point<Clock, Duration>& abs_time,
16                Predicate pred);
```

---

Все эти перегруженные варианты функции `wait` требуют предиката. Представленные версии гарантируют, что потоки будут оповещены, если запрос на прерывание для переданного `std::stop_token` был вызван. Эти функции возвращают логическое значение, говоря о том, был ли предикат равен `true`. Это возвращаемое значение не зависит от того, было запрошено прерывание или же истекло отпущенное время (`timeout`). Эти три перегруженные функции эквивалентны следующим выражениям:

Эквивалентные выражения для трех перегруженных функций

---

```
// wait in lines 1 - 4
while (!stoken.stop_requested()) {
    if (pred()) return true;
    wait(lock);
}
return pred();
```

---

<sup>1</sup> [https://en.cppreference.com/w/cpp/thread/condition\\_variable](https://en.cppreference.com/w/cpp/thread/condition_variable).

---

```

// wait_for in lines 6 - 10
return wait_until(lock,
                  std::move(stoken),
                  chrono::steady_clock::now() + rel_time,
                  std::move(pred)
                  );

// wait_until in lines 12 - 16
while (!stoken.stop_requested()) {
    if (pred()) return true;
    if (wait_until(lock, timeout_time) == std::cv_status::timeout)
        return pred();
}
return pred();

```

---

После вызова ожидания вы можете проверить, было ли запрошено прерывание.

Обработка прерывания при вызове ожидания

---

```

cv.wait(lock, stoken, predicate);
if (stoken.stop_requested()){
    // interrupt occurred
}

```

---

Следующий пример показывает, как использовать условные переменные вместе с запросом на прерывание.

Использование условной переменной для запроса прерывания

---

```

1 // conditionVariableAny.cpp
2
3 #include <condition_variable>
4 #include <thread>
5 #include <iostream>
6 #include <chrono>
7 #include <mutex>
8 #include <thread>
9
10 using namespace std::literals;
11
12 std::mutex mut;

```

```
13 std::condition_variable_any condVar;
14
15 bool dataReady;
16
17 void receiver(std::stop_token stopToken) {
18
19     std::cout << "Waiting" << '\n';
20
21     std::unique_lock<std::mutex> lck(mut);
22     bool ret = condVar.wait(lck, stopToken, []{return dataReady;});
23     if (ret){
24         std::cout << "Notification received: " << '\n';
25     }
26     else{
27         std::cout << "Stop request received" << '\n';
28     }
29 }
30
31 void sender() {
32
33     std::this_thread::sleep_for(5ms);
34     {
35         std::lock_guard<std::mutex> lck(mut);
36         dataReady = true;
37         std::cout << "Send notification" << '\n';
38     }
39     condVar.notify_one();
40
41 }
42
43 int main(){
44
45     std::cout << '\n';
46
47     std::jthread t1(receiver);
48     std::jthread t2(sender);
49
50     t1.request_stop();
51 }
```



```

52     t1.join();
53     t2.join();
54
55     std::cout << '\n';
56
57 }
```

Поток `receiver` (строки 17–29) ожидает оповещение от потока `sender` (строки 31–41). Перед тем как поток `sender` посылает свое оповещение в строке 39, главный поток запрашивает прерывание в строке 50. Вывод программы показывает, что запрос на прерывание случился перед оповещением.

```

1 // condition_variable.cpp
2
3 #include <condition_variable>
4 #include <thread>
5 #include <iostream>
6 #include <chrono>
7 #include <mutex>
8 #include <thread>
9
10 using namespace std::literals;
11
12 std::mutex mut;
13 std::condition_variable_any condvar;
14
```

Отправка запроса на прерывание через условную переменную



### Краткая информация

- ♦ Благодаря `std::stop_source`, `std::stop_token` и `std::stop_callback` потоки и условные переменные могут быть совместно использованы для кооперативного прерывания. Кооперативное прерывание означает, что поток получает запрос на прерывание, который он может принять или проигнорировать.
- ♦ Объект `std::stop_token` может быть передан операции и использован позже для того, чтобы запросить токен для запроса на прерывание или регистрации функции обратного вызова через `std::stop_callback`.
- ♦ `std::jthread` и `std::condition_variable_any` могут принимать запрос на прерывание.

## 6.6 `std::jthread`



Сиппи пытается заплести косичку

`std::jthread` обозначает `joinable thread`, т. е. поток, к которому можно присоединиться. Кроме стандартной функциональности `std::thread`<sup>1</sup> из C++11, `std::jthread` автоматически присоединяется в своем деструкторе и может быть совместно прерван.

Следующая таблица дает вам краткий обзор функциональности объекта `std::jthread t`. За более подробной информацией обратитесь к [cppreference.com](https://en.cppreference.com)<sup>2</sup>.

| Метод                                                              | Описание                                                                         |
|--------------------------------------------------------------------|----------------------------------------------------------------------------------|
| <code>t.join()</code>                                              | Ожидать до тех пор, пока поток <code>t</code> не завершит свое выполнение        |
| <code>t.detach()</code>                                            | Выполняет созданный поток <code>t</code> независимо от создателя                 |
| <code>t.joinable()</code>                                          | Возвращает <code>true</code> , если поток <code>t</code> является присоединяемым |
| <code>t.get_id()</code><br><code>std::this_thread::get_id()</code> | Возвращает идентификатор потока                                                  |
| <code>std::jthread::hardware_concurrency()</code>                  | Возвращает количество потоков, которые могут выполняться одновременно            |
| <code>std::this_thread::sleep_until(absTime)</code>                | «Усыпляет» поток <code>t</code> до момента времени <code>absTime</code>          |
| <code>std::this_thread::sleep_for(relTime)</code>                  | «Усыпляет» поток <code>t</code> на время <code>relTime</code>                    |
| <code>std::this_thread::yield()</code>                             | Позволяет системе запустить другой поток                                         |

<sup>1</sup> <https://en.cppreference.com/w/cpp/thread/thread>.

<sup>2</sup> <https://en.cppreference.com/w/cpp/thread/jthread>.

Окончание табл.

| Метод                                                     | Описание                                                                                        |
|-----------------------------------------------------------|-------------------------------------------------------------------------------------------------|
| <code>t.swap(t2)</code><br><code>std::swap(t1, t2)</code> | Меняет потоки местами                                                                           |
| <code>t.get_stop_source()</code>                          | Возвращает объект <code>std::stop_source</code> , ассоциированный с разделяемым стоп-состоянием |
| <code>t.get_stop_token()</code>                           | Возвращает объект <code>std::stop_token</code> , ассоциированный с разделяемым стоп-состоянием  |
| <code>t.request_stop()</code>                             | Запрашивает выполнение прерывания через разделяемое стоп-состояние                              |

### 6.6.1 Автоматическое присоединение

Это *неинтуитивное* поведение `std::thread`. Если `std::thread` является присоединяемым, то в его деструкторе вызывается `std::terminate`<sup>1</sup>. Поток `thr` является присоединяемым, если не вызывались ни `thr.join()`, ни `thr.detach()`.

Прерывание присоединяемого потока `std::thread`

---

```
// threadJoinable.cpp
```

```
#include <iostream>
#include <thread>

int main() {

    std::cout << '\n';
    std::cout << std::boolalpha;

    std::thread thr{[] { std::cout << "Joinable std::thread" << '\n'; }};

    std::cout << "thr.joinable(): " << thr.joinable() << '\n';

    std::cout << '\n';
}
```

---

При выполнении эта программа прерывается.

---

<sup>1</sup> <https://en.cppreference.com/w/cpp/error/terminate>.

```
// threadJoinable.cpp
```

```
#include <iostream>
```

```
#include <thread>
```

```
int main() {
```

```
    std::cout << '\n';
```

```
    std::cout << std::boolalpha;
```

```
    std::thread thr{[]{ std::cout << "Joinable std::thread" << '\n'; }};
```

```
    std::cout << "thr.joinable(): " << thr.joinable() << '\n';
```

```
    std::cout << '\n';
```

```
}
```

---

Прерывание присоединяемого потока `std::thread`

При обоих выполнениях программы происходит прерывание `std::thread`. При втором запуске у потока `thr` было достаточно времени, для того чтобы вывести сообщение "Joinable std::thread".

В следующем примере я использую `std::jthread` из текущего стандарта C++20.

Прерывание присоединяемого потока `std::thread`

---

```
// jthreadJoinable.cpp
```

```
#include <iostream>
```

```
#include <thread>
```

```
int main() {
```

```
    std::cout << '\n';
```

```
    std::cout << std::boolalpha;
```

```
    std::jthread thr{[]{ std::cout << "Joinable std::thread" << '\n'; }};
```

```
    std::cout << "thr.joinable(): " << thr.joinable() << '\n';
```

```
    std::cout << '\n';
```

```
}
```

---

Теперь поток `thr` автоматически присоединяется в своем деструкторе, если он все еще является присоединяемым.

---

```
// jthreadJoinable.cpp

#include <iostream>
#include <thread>

int main() {

    std::cout << '\n';
    std::cout << std::boolalpha;

    std::jthread thr[]{ std::cout << "Joinable std::thread" << '\n'; }};

    std::cout << "thr.joinable(): " << thr.joinable() << '\n';

    std::cout << '\n';
}
```

---

Использование автоматически присоединяемого потока `std::jthread`

Вот типичная реализация деструктора `std::jthread`.

Типичная реализация деструктора `std::jthread`

---

```
1  jthread::~jthread() {
2      if(joinable()) {
3          request_stop();
4          join();
5      }
6  }
```

---

Сначала поток проверяет, что он все еще присоединяемый (строка 2). Поток присоединяемый, если для него не вызывался ни `join()`, ни `detach()`. Если поток все еще присоединяемый, то он запрашивает приостановку выполнения (строка 3) и после этого вызывает `join()` (строка 4). Вызов `join()` блокирует поток до окончания выполнения потока.

## 6.6.2 Кооперативное прерывание `std::jthread`

Для более ясного понимания материала я приведу небольшой пример.

Прерывание непрерываемого и прерываемого `std::jthread`

---

```
1  // interruptJthread.cpp
2
3  #include <chrono>
4  #include <iostream>
5  #include <thread>
6
7  using namespace::std::literals;
8
9  int main() {
10
```

```
11     std::cout << '\n';
12
13     std::jthread nonInterruptible([]{
14         int counter{0};
15         while (counter < 10){
16             std::this_thread::sleep_for(0.2s);
17             std::cerr << "nonInterruptible: " << counter << '\n';
18             ++counter;
19         }
20     });
21
22     std::jthread interruptible([](std::stop_token token){
23         int counter{0};
24         while (counter < 10){
25             std::this_thread::sleep_for(0.2s);
26             if (token.stop_requested()) return;
27             std::cerr << "interruptible: " << counter << '\n';
28             ++counter;
29         }
30     });
31
32     std::this_thread::sleep_for(1s);
33
34     std::cerr << '\n';
35     std::cerr << "Main thread interrupts both jthreads" << '\n';
36     nonInterruptible.request_stop();
37     interruptible.request_stop();
38
39     std::cout << '\n';
40
41 }
```

---

В главной программе я запускаю два потока: `nonInterruptible` и `interruptible` (строки 13 и 22). В отличие от `nonInterruptible`, поток `interruptible` получает на вход `std::stop_token` и использует его в строке 26 для проверки того, был ли запрос на прерывание: `token.stop_requested()`. В случае такого запроса лямбда-функция возвращает управление, и поток завершается. Вызов `interruptible.request_stop()` (строка 37) вызывает запрос на прерывание. Это не справедливо для предыдущего вызова `nonInterruptible.request_stop()`. Этот вызов не оказывает никакого воздействия.

```

C:\Users\seminar>interruptJthread.exe

nonInterruptible: 0
interruptible: 0
nonInterruptible: 1
interruptible: 1
nonInterruptible: 2
interruptible: 2
nonInterruptible: 3
interruptible: 3

Main thread interrupts both jthreads

nonInterruptible: 4
nonInterruptible: 5
nonInterruptible: 6
nonInterruptible: 7
nonInterruptible: 8
nonInterruptible: 9

C:\Users\seminar>

```

Прерывание непрерываемого и прерываемого `std::jthread`



### Краткая информация

- ♦ `std::jthread` обозначает присоединяемый поток. В добавление к `std::thread` в C++11 `std::jthread` автоматически присоединяется в своем деструкторе и может быть кооперативно прерван.
- ♦ Класс `std::thread` обладает неинтуитивным поведением. Если `std::thread` по-прежнему присоединяем, то в деструкторе вызывается `std::terminate`. Для сравнения, `std::jthread` автоматически присоединяется в деструкторе при необходимости.
- ♦ `std::jthread` может быть кооперативно прерван при помощи `std::stop_token`. Кооперативно означает, что `std::jthread` может проигнорировать запрос на прерывание.

## 6.7 Синхронизированные потоки вывода



Сиппи поет в хоре



### Поддержка компиляторами синхронизированных потоков вывода

На конец 2020 года только GCC поддерживал синхронизированные потоки вывода.

Что происходит, когда вы пишете в `std::cout` без синхронизации?

Несинхронизированный доступ к `std::cout`

---

```

1  // coutUnsynchronized.cpp
2
3  #include <chrono>
4  #include <iostream>
5  #include <thread>
6
7  class Worker{
8  public:
9      Worker(std::string n):name(n) {};
10     void operator() (){
11         for (int i = 1; i <= 3; ++i) {
12             // begin work
13             std::this_thread::sleep_for(std::chrono::milliseconds(200));
14             // end work
15             std::cout << name << ": " << "Work " << i << " done !!!" << '\n';
16         }
17     }

```



---

```

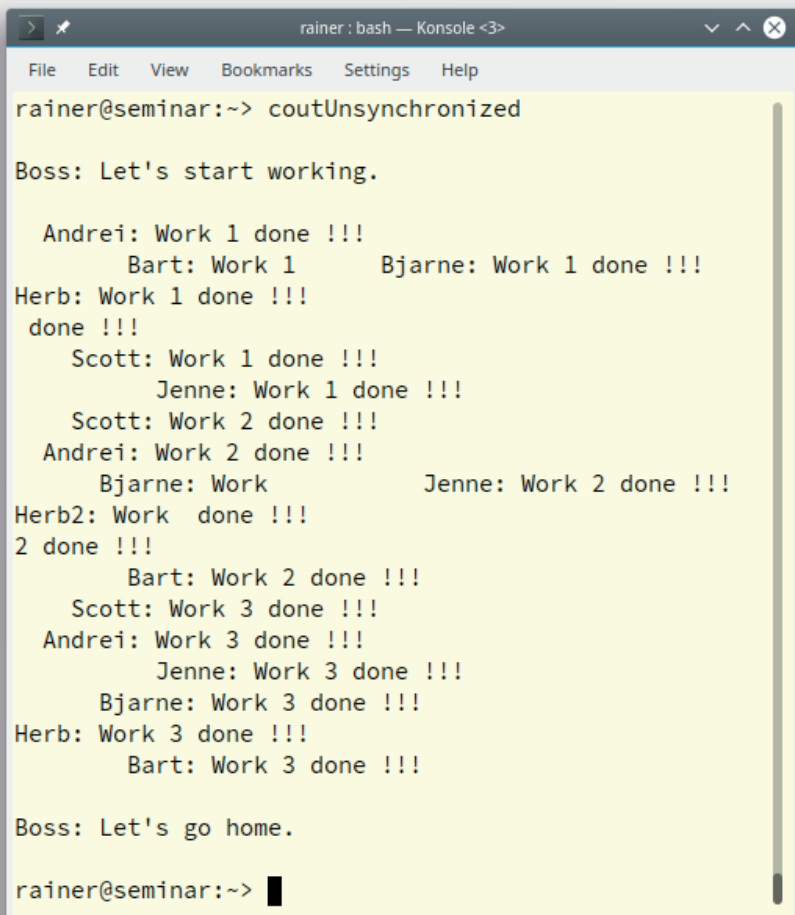
18 private:
19     std::string name;
20 };
21
22
23 int main() {
24
25     std::cout << '\n';
26
27     std::cout << "Boss: Let's start working.\n\n";
28
29     std::thread herb= std::thread(Worker("Herb"));
30     std::thread andrei= std::thread(Worker(" Andrei"));
31     std::thread scott= std::thread(Worker(" Scott"));
32     std::thread bjarne= std::thread(Worker(" Bjarne"));
33     std::thread bart= std::thread(Worker(" Bart"));
34     std::thread jenne= std::thread(Worker(" Jenne"));
35
36
37     herb.join();
38     andrei.join();
39     scott.join();
40     bjarne.join();
41     bart.join();
42     jenne.join();
43
44     std::cout << "\n" << "Boss: Let's go home." << '\n';
45
46     std::cout << '\n';
47
48 }

```

---

Здесь есть шесть рабочих потоков (строки 29–34). Каждый рабочий поток выполняет три блока работы, каждый из которых занимает 1/5 секунды (строка 13). После выполнения каждого блока поток генерирует сообщение об этом (строка 15). После того как главный поток получит уведомления от всех шестерых работников, он возвращает их (строка 44).

Такой простой сценарий приводит к бардаку: каждый поток будет выводить свое сообщение, игнорируя при этом все остальные.



```

rainer : bash — Konsole <3>
File Edit View Bookmarks Settings Help

rainer@seminar:~> coutUnsynchronized

Boss: Let's start working.

  Andrei: Work 1 done !!!
      Bart: Work 1      Bjarne: Work 1 done !!!
Herb: Work 1 done !!!
done !!!
  Scott: Work 1 done !!!
      Jenne: Work 1 done !!!
  Scott: Work 2 done !!!
  Andrei: Work 2 done !!!
      Bjarne: Work      Jenne: Work 2 done !!!
Herb2: Work done !!!
2 done !!!
      Bart: Work 2 done !!!
  Scott: Work 3 done !!!
  Andrei: Work 3 done !!!
      Jenne: Work 3 done !!!
      Bjarne: Work 3 done !!!
Herb: Work 3 done !!!
      Bart: Work 3 done !!!

Boss: Let's go home.

rainer@seminar:~> █

```

Несинхронизированный вывод в `std::cout`



### **`std::cout` является потокобезопасным**

Стандарт C++11 гарантирует, что вам не нужно защищать `std::cout`. Каждый символ пишется атомарно. Но при этом несколько операций вывода, как в примере выше, могут привести к перемешиванию результатов. Подобное перемешивание – это только визуальная проблема: программа является хорошо определенной (well-defined). Это справедливо для всех глобальных потоков. Добавление и чтение из глобальных потоков (`std::cout`, `std::cin`, `std::cerr` и `std::clog`) являются потокобезопасными. Говоря более формально: запись в `std::cout` не участвует в гонке данных (data race), но создает состояние гонки (race condition). Это значит, что выход зависит от «переплетения» потоков.

Как мы можем решить эту проблему? В C++11 ответ прост: использовать блокировку, такую как `lock_guard`<sup>1</sup>, для синхронизации доступа к `std::cout`.

Синхронизированный доступ к `std::cout`

---

```

1  // coutSynchronized.cpp
2
3  #include <chrono>
4  #include <iostream>
5  #include <mutex>
6  #include <thread>
7
8  std::mutex coutMutex;
9
10 class Worker{
11 public:
12     Worker(std::string n):name(n) {};
13
14     void operator() () {
15         for (int i = 1; i <= 3; ++i) {
16             // begin work
17             std::this_thread::sleep_for(std::chrono::milliseconds(200));
18             // end work
19             std::lock_guard<std::mutex> coutLock(coutMutex);
20             std::cout << name << ": " << "Work " << i << " done !!!\n";
21         }
22     }
23 private:
24     std::string name;
25 };
26
27
28 int main() {
29
30     std::cout << '\n';
31
32     std::cout << "Boss: Let's start working." << "\n\n";
33
34     std::thread herb= std::thread(Worker("Herb"));
```

---

<sup>1</sup> [https://en.cppreference.com/w/cpp/thread/lock\\_guard](https://en.cppreference.com/w/cpp/thread/lock_guard).

```
35  std::thread andrei= std::thread(Worker("  Andrei"));
36  std::thread scott= std::thread(Worker("    Scott"));
37  std::thread bjarne= std::thread(Worker("      Bjarne"));
38  std::thread bart= std::thread(Worker("        Bart"));
39  std::thread jenne= std::thread(Worker("          Jenne"));
40
41  herb.join();
42  andrei.join();
43  scott.join();
44  bjarne.join();
45  bart.join();
46  jenne.join();
47
48  std::cout << "\n" << "Boss: Let's go home." << '\n';
49
50  std::cout << '\n';
51
52 }
```

---

`coutMutex` в строке 8 защищает разделяемый объект `std::cout`. Помещение `coutMutex` в `std::lock_guard` гарантирует, что `coutMutex` будет заблокирован в конструкторе (строке 19) и разблокирован в деструкторе (строка 21) `std::lock_guard`. Благодаря `coutMutex` и `coutLock` вместо беспорядка наступает гармония.

```

rainer : bash — Konsole <3>
File Edit View Bookmarks Settings Help

rainer@seminar:~> coutSynchronized

Boss: Let's start working.

    Scott: Work 1 done !!!
    Bjarne: Work 1 done !!!
    Andrei: Work 1 done !!!
    Herb: Work 1 done !!!
        Jenne: Work 1 done !!!
        Bart: Work 1 done !!!
    Scott: Work 2 done !!!
    Andrei: Work 2 done !!!
        Bjarne: Work 2 done !!!
    Herb: Work 2 done !!!
        Bart: Work 2 done !!!
        Jenne: Work 2 done !!!
    Andrei: Work 3 done !!!
        Scott: Work 3 done !!!
        Bjarne: Work 3 done !!!
    Herb: Work 3 done !!!
        Bart: Work 3 done !!!
        Jenne: Work 3 done !!!

Boss: Let's go home.

rainer@seminar:~>

```

Синхронизированный доступ к `std::cout`

С публикацией стандарта C++20 синхронизированный вывод в `std::cout` стал очень простым. Класс `std::basic_syncbuf` – это обертка для `std::basic_streambuf`<sup>1</sup>. Он собирает вывод в буфере. Эта обертка устанавливает обернутый буфер в качестве своего содержимого при уничтожении. Соответственно, содержимое выводится как непрерывная последовательность символов, иначе происходит перемешивание с выводом других потоков.

Благодаря `std::basic_ostream` вы можете непосредственно писать в `std::cout`.

<sup>1</sup> [https://en.cppreference.com/w/cpp/io/basic\\_streambuf](https://en.cppreference.com/w/cpp/io/basic_streambuf).

Вы можете создать именованный синхронизированный выходной поток. Ниже приводится предыдущий пример, преобразованный для использования синхронизированного `std::cout`.

Синхронизированный доступ к `std::cout` через `std::basic_ostream`

---

```
1  // synchronizedOutput.cpp
2
3  #include <chrono>
4  #include <iostream>
5  #include <syncstream>
6  #include <thread>
7
8  class Worker{
9  public:
10     Worker(std::string n): name(n) {};
11     void operator() (){
12         for (int i = 1; i <= 3; ++i) {
13             // begin work
14             std::this_thread::sleep_for(std::chrono::milliseconds(200));
15             // end work
16             std::osyncstream syncStream(std::cout);
17             syncStream << name << ": " << "Work " << i << " done !!!" << '\n';
18         }
19     }
20 private:
21     std::string name;
22 };
23
24
25 int main() {
26
27     std::cout << '\n';
28
29     std::cout << "Boss: Let's start working.\n\n";
30
31     std::thread herb= std::thread(Worker("Herb"));
32     std::thread andrei= std::thread(Worker("  Andrei"));
33     std::thread scott= std::thread(Worker("    Scott"));
34     std::thread bjarne= std::thread(Worker("      Bjarne"));
35     std::thread bart= std::thread(Worker("        Bart"));
36     std::thread jenne= std::thread(Worker("          Jenne"));
```

```

37
38
39     herb.join();
40     andrei.join();
41     scott.join();
42     bjarne.join();
43     bart.join();
44     jenne.join();
45
46     std::cout << "\n" << "Boss: Let's go home." << '\n';
47
48     std::cout << '\n';
49
50 }

```

Единственным отличием от предыдущей программы является то, что мы заворачиваем `std::cout` в `std::osyncstream` (строка 16). Для использования `std::osyncstream` я подключил заголовочный файл `<syncstream>`. Когда `std::osyncstream` выходит из зоны видимости в строке 18, то символы перемещаются и `std::cout` опустошает свой буфер (flush). Стоит упомянуть, что вызовы `std::cout` в главном потоке не приводят к состоянию гонки и поэтому не нуждаются в синхронизации.

Поскольку я всего лишь один раз использую объявленный в строке 17 `syncStream`, то более разумно будет использовать какой-нибудь временный объект. Следующий пример кода показывает измененный оператор вызова.

```

void operator>() {
    for (int i = 1; i <= 3; ++i) {
        // begin work
        std::this_thread::sleep_for(std::chrono::milliseconds(200));
        // end work
        std::osyncstream(std::cout) << name << ": " << "Work " << i << " done !!!"
                                   << '\n';
    }
}

```

`std::osyncstream syncStream` имеет два интересных метода:

- `syncStream.emit()` выводит весь буферизованный вывод и выполняет сброс буферов;
- `syncStream.get_wrapped()` возвращает указатель на «обернутый» буфер.

Сайт [cppreference.com](https://en.cppreference.com)<sup>1</sup> показывает, как вы можете использовать метод `get_wrapped`.

<sup>1</sup> [https://en.cppreference.com/w/cpp/io/basic\\_osyncstream/get\\_wrapped](https://en.cppreference.com/w/cpp/io/basic_osyncstream/get_wrapped).

Последовательный вывод

---

```
// sequenceOutput.cpp
```

```
#include <syncstream>
#include <iostream>
int main() {

    std::osyncstream bout1(std::cout);
    bout1 << "Hello, ";
    {
        std::osyncstream(bout1.get_wrapped()) << "Goodbye, " << "Planet!" << '\n';
    } // emits the contents of the temporary buffer

    bout1 << "World!" << '\n';

} // emits the contents of bout1
```

---

```
// sequenceOutput.cpp
#include <syncstream>
#include <iostream>
int main() {

    std::osyncstream bout1(std::cout);
    bout1 << "Hello, ";
    {
        std::osyncstream(bout1.get_wrapped()) << "Goodbye, " << "Planet!" << '\n';
    }
}
```

Синхронизированный доступ к `std::cout`



### Краткая информация

- ♦ Хотя `std::cout` и является потокобезопасным, вы можете получить перемешанный вывод, в случае когда потоки одновременно пишут в `std::cout`. Это выглядит не очень хорошо, но не является критичным и не является гонкой данных.
- ♦ В C++20 поддерживаются синхронизированные выходные потоки. Они накапливают вывод во внутреннем буфере и затем атомарно записывают его содержимое. Соответственно, при этом не происходит перемешивания вывода с другими потоками.

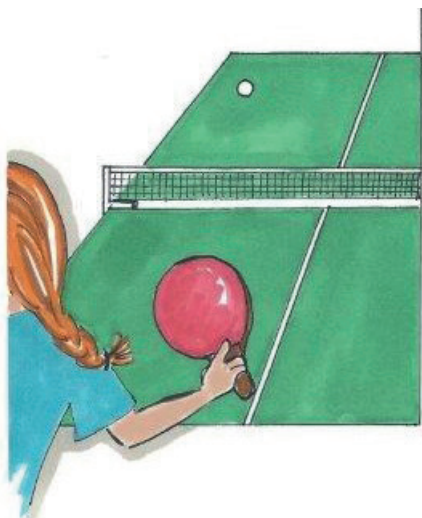


## 7. Практические примеры

После того как вы ознакомились с теорией по C++20, применим эту теорию на практике. Я покажу вам несколько практических примеров (case studies).

Когда вы хотите синхронизировать потоки более одного раза, вы можете использовать условные переменные, `std::atomic_flag`, `std::atomic<bool>` или семафоры. В разделе «Быстрая синхронизация потоков» я хочу ответить на вопрос, что из этого быстрее. Раздел по сопрограммам покажет три сопрограммы, основанные на `co_return`, `co_yield` и `co_await`. Я использую эти сопрограммы как отправную точку для дальнейших экспериментов для углубления нашего понимания архитектуры использования сопрограмм. В разделе «Вариации объектов `future`» я реализую ленивый объект `future` и объект `future`, основанный на объекте, который был у нас в разделе по `co_return`. В разделе «Изменения и обобщение потоков» я улучшу генератор из раздела по `co_return`. А в разделе «Различные архитектуры, основанные на заданиях» будет обсуждаться архитектура с задачами (jobs), рассмотрение которой было начато в разделе по `co_await`.

## 7.1 Быстрая синхронизация потоков



Сиппи играет в пинг-понг



### Компьютеры, используемые для сравнения

Вы должны понимать, что приводимое быстродействие измеряется для конкретного компьютера. Имеет смысл сравнивать относительное быстродействие для одного и того же компьютера. Я бы не хотел сравнивать быстродействие моего настольного компьютера под управлением Linux и моего ноутбука под управлением Windows, но мне интересно, какие алгоритмы лучше работают под Windows, а какие под Linux.

Когда вы хотите синхронизировать потоки более одного раза, то вы можете использовать условные переменные, `std::atomic_flag`, `std::atomic<bool>` или семафоры. В этом разделе я отвечу на вопрос, какой из них самый быстрый.

Для получения данных для сравнения я собираюсь реализовать игру в пинг-понг. Один поток выполняет функцию `ping`, а другой – функцию `pong`. Поток `ping` ждет уведомления от `pong`-потока и посылает ему уведомление. Игра останавливается после 1 000 000 таких переходов. Я запускаю игру пять раз для получения сравнимых значений быстродействия.



### О данных

Я выполнял свои тесты быстродействия в конце 2020 года с новым компилятором Visual Studio 19.28, поскольку он уже поддерживал синхронизацию при помощи атомиков (`std::atomic_flag` и `std::atomic`) и семафоров. Кроме того, я компилировал эти примеры с максимальной оптимизацией (`/Ox`). Полученные данные позволяют дать приблизительную оценку относительного быстродействия различных способов синхронизации потоков. Если вам нужны данные для вашей платформы, просто повторите тесты на ней.

Начнем со сравнения с C++11.

## 7.1.1 Условные переменные

Многократная синхронизация при помощи условных переменных

---

```

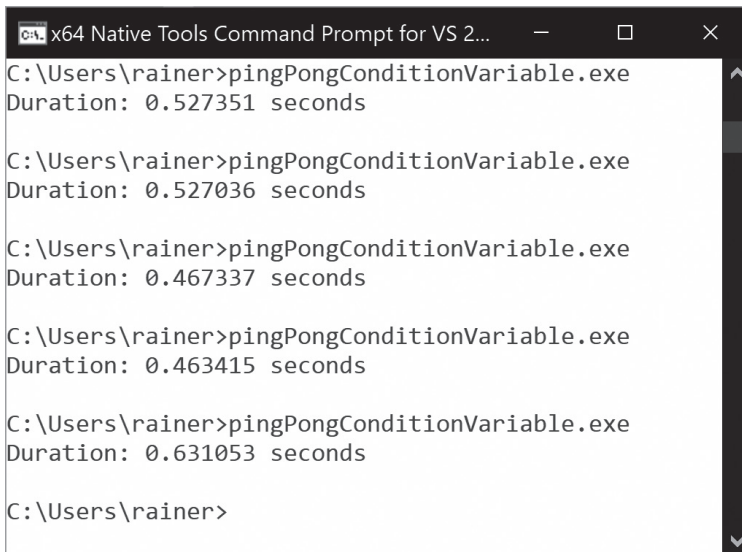
1  // pingPongConditionVariable.cpp
2
3  #include <condition_variable>
4  #include <iostream>
5  #include <atomic>
6  #include <thread>
7
8  bool dataReady{false};
9
10 std::mutex mutex_;
11 std::condition_variable condVar1;
12 std::condition_variable condVar2;
13
14 std::atomic<int> counter{};
15 constexpr int countlimit = 1'000'000;
16
17 void ping() {
18
19     while(counter <= countlimit) {
20         {
21             std::unique_lock<std::mutex> lck(mutex_);
22             condVar1.wait(lck, []{return dataReady == false;});
23             dataReady = true;
24         }
25         ++counter;
26         condVar2.notify_one();
27     }
28 }
29
30 void pong() {
31
32     while(counter < countlimit) {
33         {
34             std::unique_lock<std::mutex> lck(mutex_);
35             condVar2.wait(lck, []{return dataReady == true;});
36             dataReady = false;
37         }
38         condVar1.notify_one();
39     }
40
41 }
```

```
42
43 int main(){
44
45     auto start = std::chrono::system_clock::now();
46
47     std::thread t1(ping);
48     std::thread t2(pong);
49
50     t1.join();
51     t2.join();
52
53     std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
54     std::cout << "Duration: " << dur.count() << " seconds" << '\n';
55 }
```

---

Я использую в этой программе две условные переменные – `condVar1` и `condVar2`. Поток `ping` ожидает оповещения от условной переменной `condVar1` и отправляет свое оповещение через переменную `condVar2`. Переменная `dataReady` защищает от ложных и потерянных пробуждений. Игра пинг-понг заканчивается, когда `counter` достигает `countlimit`. Вызовы `notify_one` (строки 26 и 38) и `counter` являются потокобезопасными и поэтому расположены вне критической области.

Вот полученная статистика.



```

C:\Users\rainer>pingPongConditionVariable.exe
Duration: 0.527351 seconds

C:\Users\rainer>pingPongConditionVariable.exe
Duration: 0.527036 seconds

C:\Users\rainer>pingPongConditionVariable.exe
Duration: 0.467337 seconds

C:\Users\rainer>pingPongConditionVariable.exe
Duration: 0.463415 seconds

C:\Users\rainer>pingPongConditionVariable.exe
Duration: 0.631053 seconds

C:\Users\rainer>
```

Многочисленная синхронизация при помощи условных переменных

Среднее время выполнения 0,52 с.

Перенос этого кода на `std::atomic_flag` довольно прост.

## 7.1.2 std::atomic\_flag

Вот тот же самый код, использующий два атомарных флага и потом всего один флаг.

### 7.1.2.1 Два атомарных флага

В следующей программе я заменил ожидание условной переменной на ожидание атомарного флага и оповещение при помощи условной переменной на установку атомарного флага, за которым следует оповещение.

Многократная синхронизация при помощи двух атомарных флагов

---

```

1  // pingPongAtomicFlags.cpp
2
3  #include <iostream>
4  #include <atomic>
5  #include <thread>
6
7  std::atomic_flag condAtomicFlag1{};
8  std::atomic_flag condAtomicFlag2{};
9
10 std::atomic<int> counter{};
11 constexpr int countlimit = 1'000'000;
12
13 void ping() {
14     while(counter <= countlimit) {
15         condAtomicFlag1.wait(false);
16         condAtomicFlag1.clear();
17
18         ++counter;
19
20         condAtomicFlag2.test_and_set();
21         condAtomicFlag2.notify_one();
22     }
23 }
24
25 void pong() {
26     while(counter < countlimit) {
27         condAtomicFlag2.wait(false);
28         condAtomicFlag2.clear();
29
30         condAtomicFlag1.test_and_set();
31         condAtomicFlag1.notify_one();
32     }
33 }
34
35 int main() {
36
```

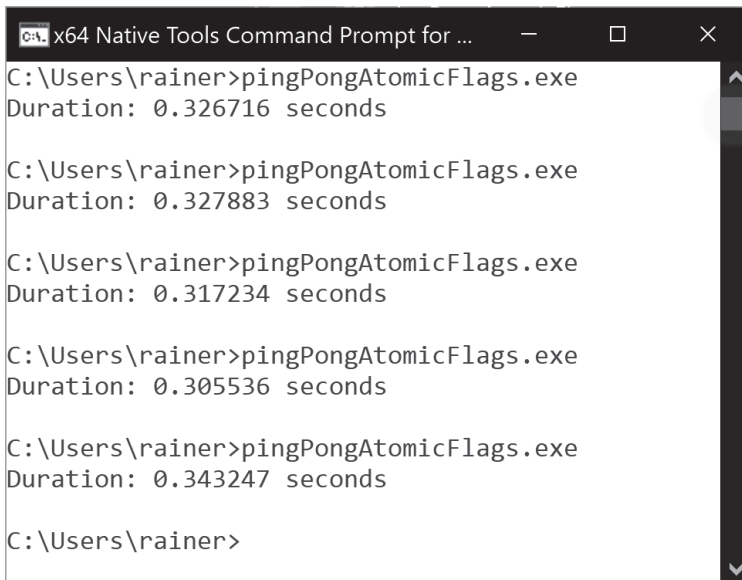
```

37  auto start = std::chrono::system_clock::now();
38
39  condAtomicFlag1.test_and_set();
40  std::thread t1(ping);
41  std::thread t2(pong);
42
43  t1.join();
44  t2.join();
45
46  std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
47  std::cout << "Duration: " << dur.count() << " seconds" << '\n';
48
49  }

```

Вызов `condAtomicFlag1.wait(false)` (строка 15) блокируется, если значение атомарного флага равно `false`, и возвращает управление, если значение `condAtomicFlag1` равно `true`. Логическое значение выполняет роль предиката и поэтому должно быть установлено в `false` (строка 15). Прежде чем уведомление (строка 21) посылается `pong`-потoku, значение `condAtomicFlag1` устанавливается в `true` (строка 20). Начальная установка `condAtomicFlag1` в `true` (строка 39) начинает игру.

Благодаря `std::atomic_flag` игра заканчивается быстрее.



```

C:\Users\rainer>pingPongAtomicFlags.exe
Duration: 0.326716 seconds

C:\Users\rainer>pingPongAtomicFlags.exe
Duration: 0.327883 seconds

C:\Users\rainer>pingPongAtomicFlags.exe
Duration: 0.317234 seconds

C:\Users\rainer>pingPongAtomicFlags.exe
Duration: 0.305536 seconds

C:\Users\rainer>pingPongAtomicFlags.exe
Duration: 0.343247 seconds

C:\Users\rainer>

```

Многочастная синхронизация при помощи двух атомарных флагов

В среднем игра занимает 0,32 с.

В ходе анализа программы можно заметить, что на самом деле может быть достаточно всего одного атомарного флага.

### 7.1.2.2 Один атомарный флаг

Использование одного атомарного флага делает архитектуру более понятной.

Многократная синхронизация при помощи одного атомарного флага

---

```

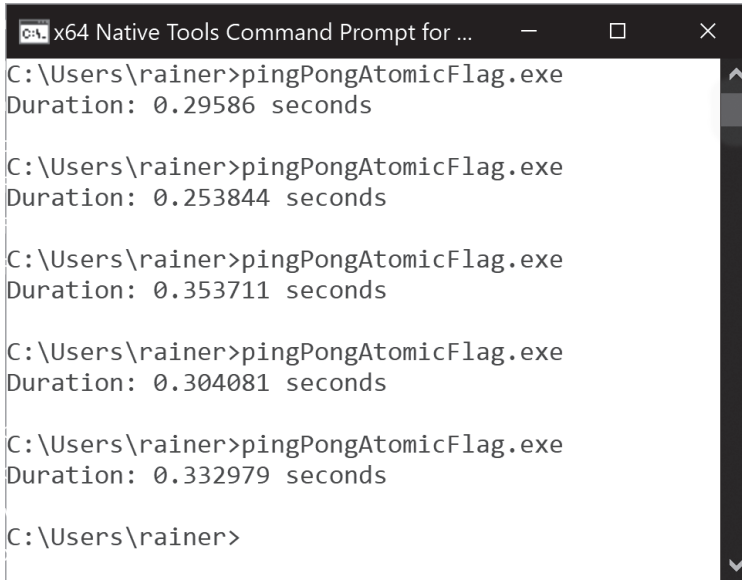
1  // pingPongAtomicFlag.cpp
2
3  #include <iostream>
4  #include <atomic>
5  #include <thread>
6
7  std::atomic_flag condAtomicFlag{};
8
9  std::atomic<int> counter{};
10 constexpr int countlimit = 1'000'000;
11
12 void ping() {
13     while(counter <= countlimit) {
14         condAtomicFlag.wait(true);
15         condAtomicFlag.test_and_set();
16
17         ++counter;
18
19         condAtomicFlag.notify_one();
20     }
21 }
22
23 void pong() {
24     while(counter < countlimit) {
25         condAtomicFlag.wait(false);
26         condAtomicFlag.clear();
27         condAtomicFlag.notify_one();
28     }
29 }
30
31 int main() {
32
33     auto start = std::chrono::system_clock::now();
34
35     condAtomicFlag.test_and_set();
36     std::thread t1(ping);
37     std::thread t2(pong);
38
39     t1.join();
40     t2.join();
41

```

```
42  std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
43  std::cout << "Duration: " << dur.count() << " seconds" << '\n';
44
45 }
```

---

В этом случае поток ping блокируется на true, а поток pong блокируется на false. С точки зрения быстродействия не имеет значения, использовать один или два атомарных флага.



```
C:\Users\rainer>pingPongAtomicFlag.exe
Duration: 0.29586 seconds

C:\Users\rainer>pingPongAtomicFlag.exe
Duration: 0.253844 seconds

C:\Users\rainer>pingPongAtomicFlag.exe
Duration: 0.353711 seconds

C:\Users\rainer>pingPongAtomicFlag.exe
Duration: 0.304081 seconds

C:\Users\rainer>pingPongAtomicFlag.exe
Duration: 0.332979 seconds

C:\Users\rainer>
```

Многократная синхронизация при помощи одного атомарного флага

Среднее время выполнения в этом случае равно 0,31 с.

В этом примере я использовал `std::atomic_flag` как атомарное логическое значение. Давайте теперь попробуем использовать `std::atomic<bool>`.

### 7.1.3 `std::atomic<bool>`

Следующая реализация на C++20 основана на `std::atomic`.

Многоразовая синхронизация при помощи атомарного логического значения

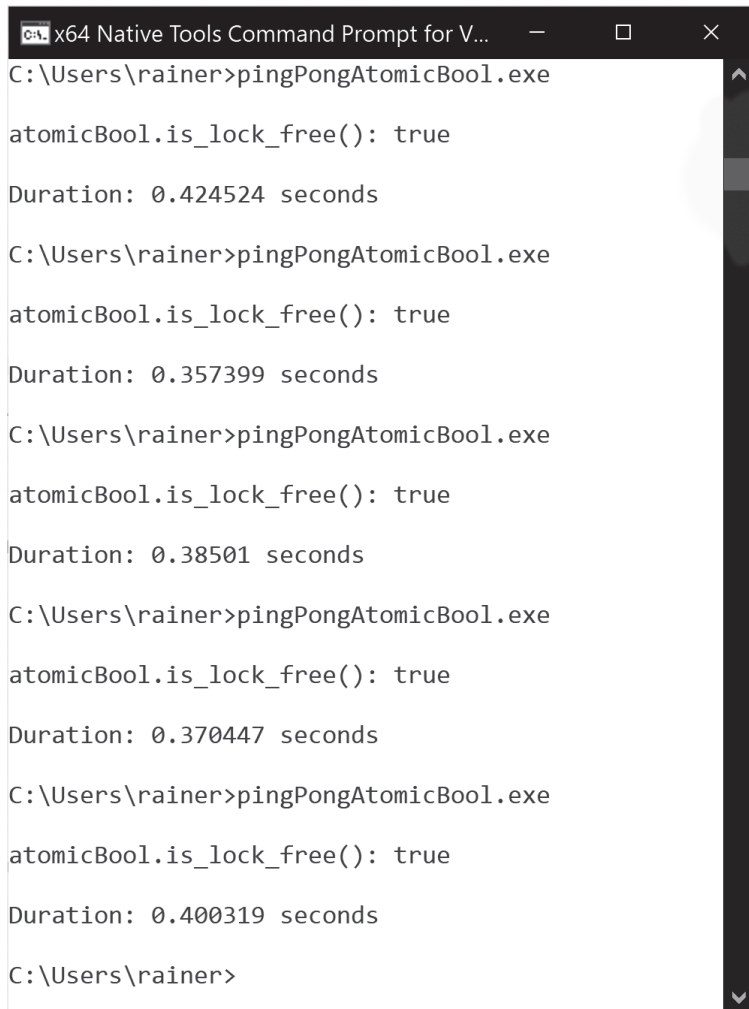
---

```
1  // pingPongAtomicBool.cpp
2
3  #include <iostream>
4  #include <atomic>
5  #include <thread>
6
7  std::atomic<bool> atomicBool{};
8
```



```
9 std::atomic<int> counter{};
10 constexpr int countlimit = 1'000'000;
11
12 void ping() {
13     while(counter <= countlimit) {
14         atomicBool.wait(true);
15         atomicBool.store(true);
16
17         ++counter;
18
19         atomicBool.notify_one();
20     }
21 }
22
23 void pong() {
24     while(counter < countlimit) {
25         atomicBool.wait(false);
26         atomicBool.store(false);
27         atomicBool.notify_one();
28     }
29 }
30
31 int main() {
32
33     std::cout << std::boolalpha << '\n';
34
35     std::cout << "atomicBool.is_lock_free(): "
36                 << atomicBool.is_lock_free() << '\n';
37
38     std::cout << '\n';
39
40     auto start = std::chrono::system_clock::now();
41
42     atomicBool.store(true);
43     std::thread t1(ping);
44     std::thread t2(pong);
45
46     t1.join();
47     t2.join();
48
49     std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
50     std::cout << "Duration: " << dur.count() << " seconds" << '\n';
51
52 }
```

Класс `std::atomic<bool>` внутри себя может использовать механизм блокировки, например, мьютекс. Моя реализация под Windows лишена блокировок.



```
C:\Users\rainer>pingPongAtomicBool.exe
atomicBool.is_lock_free(): true
Duration: 0.424524 seconds

C:\Users\rainer>pingPongAtomicBool.exe
atomicBool.is_lock_free(): true
Duration: 0.357399 seconds

C:\Users\rainer>pingPongAtomicBool.exe
atomicBool.is_lock_free(): true
Duration: 0.38501 seconds

C:\Users\rainer>pingPongAtomicBool.exe
atomicBool.is_lock_free(): true
Duration: 0.370447 seconds

C:\Users\rainer>pingPongAtomicBool.exe
atomicBool.is_lock_free(): true
Duration: 0.400319 seconds

C:\Users\rainer>
```

Многоразовая синхронизация при помощи атомарного логического значения

В среднем время выполнения составляет 0,38 с.

С точки зрения читаемости эта реализация, основанная на `std::atomic`, очень проста для понимания. Это также справедливо и для следующей реализации, основанной на семафорах.

### 7.1.4 Семафоры

Семафоры обещают быть быстрее условных переменных. Давайте проверим, так ли это.

## Многоразовая синхронизация при помощи семафоров

---

```

1  // pingPongSemaphore.cpp
2
3  #include <iostream>
4  #include <semaphore>
5  #include <thread>
6
7  std::counting_semaphore<1> signal2Ping(0);
8  std::counting_semaphore<1> signal2Pong(0);
9
10 std::atomic<int> counter{};
11 constexpr int countlimit = 1'000'000;
12
13 void ping() {
14     while(counter <= countlimit) {
15         signal2Ping.acquire();
16         ++counter;
17         signal2Pong.release();
18     }
19 }
20
21 void pong() {
22     while(counter < countlimit) {
23         signal2Pong.acquire();
24         signal2Ping.release();
25     }
26 }
27
28 int main() {
29
30     auto start = std::chrono::system_clock::now();
31
32     signal2Ping.release();
33     std::thread t1(ping);
34     std::thread t2(pong);
35
36     t1.join();
37     t2.join();
38
39     std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
40     std::cout << "Duration: " << dur.count() << " seconds" << '\n';
41
42 }
```

---

Программа `pingPongSemaphore.cpp` использует два семафора – `signal2Ping` и `signal2Pong` (строки 7 и 8). Оба могут принимать всего два значения – 0 и 1 – и инициализируются 0. Это значит, что когда значение равно 0 для семафора `signal2Ping`, то вызов `signal2Ping.release()` (строки 24 и 32) устанавливает его значение в 1 и таким образом посылает оповещение. Вызов `signal2Ping.acquire()` (строка 15) блокируется до тех пор, пока значение не станет равным 1. То же самое справедливо и для второго семафора `signal2Pong`.

```

C:\Users\rainer>pingPongSemaphore.exe
Duration: 0.367456 seconds

C:\Users\rainer>pingPongSemaphore.exe
Duration: 0.359944 seconds

C:\Users\rainer>pingPongSemaphore.exe
Duration: 0.339582 seconds

C:\Users\rainer>pingPongSemaphore.exe
Duration: 0.308024 seconds

C:\Users\rainer>pingPongSemaphore.exe
Duration: 0.319354 seconds

C:\Users\rainer>
    
```

Многоразовая синхронизация при помощи семафоров

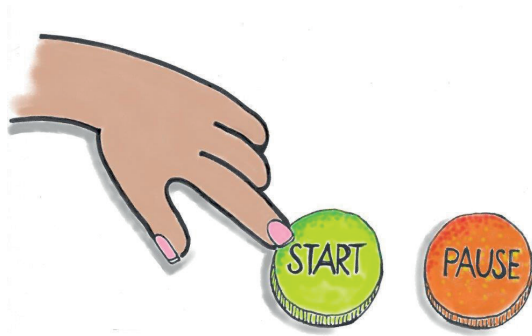
В среднем время выполнения этой программы равно 0,33 с.

### 7.1.5 Общая статистика

Как и ожидалось, условные переменные – это самый медленный вариант для синхронизации потоков, а атомарные флаги – самый быстрый. Быстродействие `std::atomic<bool>` находится где-то посередине. У использования `std::atomic<bool>` есть один недостаток. `std::atomic_flag` – это единственный атомарный тип данных, лишенный блокировок (lock-free). Больше всего меня удивили семафоры, поскольку они практически так же быстры, как и атомарные флаги.

|                               | Условные<br>переменные | Два атомар-<br>ных флага | Один<br>атомарный<br>флаг | Атомарное<br>логическое<br>значение | Семафоры |
|-------------------------------|------------------------|--------------------------|---------------------------|-------------------------------------|----------|
| <b>Время вы-<br/>полнения</b> | 0,52                   | 0,32                     | 0,31                      | 0,38                                | 0,33     |

## 7.2 Вариации объектов future



Сиппи запускает процесс

Прежде чем я буду создавать варианты объекта future из раздела про `co_return`, нам нужно понять, как будет происходить его выполнение. Комментарии делают поток выполнения более ясным. Кроме того, я даю ссылку на версию программы для онлайн-компиляторов.

Выполнение для «жадного» объекта future

---

```

1  // eagerFutureWithComments.cpp
2
3  #include <coroutine>
4  #include <iostream>
5  #include <memory>
6
7  template<typename T>
8  struct MyFuture {
9      std::shared_ptr<T> value;
10     MyFuture(std::shared_ptr<T> p): value(p) {
11         std::cout << "MyFuture::MyFuture" << '\n';
12     }
13     ~MyFuture() {
14         std::cout << "MyFuture::~MyFuture" << '\n';
15     }
16     T get() {
17         std::cout << "MyFuture::get" << '\n';
18         return *value;
19     }
20 
```

```
21  struct promise_type {
22      std::shared_ptr<T> ptr = std::make_shared<T>();
23      promise_type() {
24          std::cout << " promise_type::promise_type" << '\n';
25      }
26      ~promise_type() {
27          std::cout << " promise_type::~~promise_type" << '\n';
28      }
29      MyFuture<T> get_return_object() {
30          std::cout << " promise_type::get_return_object" << '\n';
31          return ptr;
32      }
33      void return_value(T v) {
34          std::cout << " promise_type::return_value" << '\n';
35          *ptr = v;
36      }
37      std::suspend_never initial_suspend() {
38          std::cout << " promise_type::initial_suspend" << '\n';
39          return {};
40      }
41      std::suspend_never final_suspend() noexcept {
42          std::cout << " promise_type::final_suspend" << '\n';
43          return {};
44      }
45      void unhandled_exception() {
46          std::exit(1);
47      }
48  };
49  };
50
51  MyFuture<int> createFuture() {
52      std::cout << "createFuture" << '\n';
53      co_return 2021;
54  }
55
56  int main() {
57
58      std::cout << '\n';
59  }
```

```

60     auto fut = createFuture();
61     auto res = fut.get();
62     std::cout << "res: " << res << '\n';
63
64     std::cout << '\n';
65
66 }

```

Вызов `createFuture` (строка 60) осуществляет создание экземпляра класса `MyFuture` (строка 59). Перед тем как вызов конструктора `MyFuture` (строка 10) завершится, создается, выполняется и уничтожается объект-обещание `promise_type` (строки 20–48). Объект-обещание на каждом шагу использует `std::suspend_never` (строки 36 и 40) и поэтому никогда не приостанавливает выполнение. Чтобы сохранить результат обещания для будущего вызова функции `fut.get()` (строка 60), следует выделить память / зарезервировать память / создать объект. Более того, использование `std::shared_ptr` гарантирует (строки 9 и 21), что программа не приведет к утечке памяти. Локальная переменная `fut` выходит из области видимости в строке 65, и C++ автоматически вызывает его деструктор.

Вы можете попробовать скомпилировать эту программу при помощи `Compiler Explorer`<sup>1</sup>.

```

1 // eagerFutureWithComments.cpp
2
3 #include <coroutine>
4 #include <iostream>
5 #include <memory>
6
7 template<typename T>
8 struct MyFuture {
9     std::shared_ptr<T> value;
10    MyFuture(std::shared_ptr<T> p): value(p) {
11        std::cout << "MyFuture::MyFuture" << '\n';
12    }
13    ~MyFuture() {
14        std::cout << "MyFuture::~MyFuture" << '\n';
15    }
16    T get() {
17        std::cout << "MyFuture::get" << '\n';
18        return *value;
19    }
20
21    struct promise_type {
22        std::shared_ptr<T> ptr = std::make_shared<T>();
23        promise_type() {
24            std::cout << "promise_type::promise_type" << '\n';
25        }
26        ~promise_type() {

```

«Жадный» объект future

Представленная сопрограмма выполняется сразу и поэтому является жадной (*eager*). Более того, сопрограмма выполняется в вызывающем потоке.

Давайте теперь сделаем нашу сопрограмму отложенной (*lazy*).

<sup>1</sup> <https://godbolt.org/z/Y9naEx>.

### 7.2.1 Ленивый объект future

Ленивый объект future – это такой объект future, который выполняется, только если у него запросили его значение. Давайте посмотрим, что нужно изменить в жадной сопрограмме, показанной ранее, чтобы превратить ее в ленивую.

Ленивый объект future

---

```
1  // lazyFuture.cpp
2
3  #include <coroutine>
4  #include <iostream>
5  #include <memory>
6
7  template<typename T>
8  struct MyFuture {
9      struct promise_type;
10     using handle_type = std::coroutine_handle<promise_type>;
11
12     handle_type coro;
13
14     MyFuture(handle_type h): coro(h) {
15         std::cout << "MyFuture::MyFuture" << '\n';
16     }
17     ~MyFuture() {
18         std::cout << "MyFuture::~MyFuture" << '\n';
19         if ( coro ) coro.destroy();
20     }
21
22     T get() {
23         std::cout << "MyFuture::get" << '\n';
24         coro.resume();
25         return coro.promise().result;
26     }
27
28     struct promise_type {
29         T result;
30         promise_type() {
31             std::cout << "promise_type::promise_type" << '\n';
32         }
33         ~promise_type() {
34             std::cout << "promise_type::~promise_type" << '\n';
35         }
36     }
```



---

```

36     auto get_return_object() {
37         std::cout << " promise_type::get_return_object" << '\n';
38         return MyFuture{handle_type::from_promise(*this)};
39     }
40     void return_value(T v) {
41         std::cout << " promise_type::return_value" << '\n';
42         result = v;
43     }
44     std::suspend_always initial_suspend() {
45         std::cout << " promise_type::initial_suspend" << '\n';
46         return {};
47     }
48     std::suspend_always final_suspend() noexcept {
49         std::cout << " promise_type::final_suspend" << '\n';
50         return {};
51     }
52     void unhandled_exception() {
53         std::exit(1);
54     }
55 };
56 };
57
58 MyFuture<int> createFuture() {
59     std::cout << "createFuture" << '\n';
60     co_return 2021;
61 }
62
63 int main() {
64
65     std::cout << '\n';
66
67     auto fut = createFuture();
68     auto res = fut.get();
69     std::cout << "res: " << res << '\n';
70
71     std::cout << '\n';
72
73 }

```

---

Изучим сначала наш объект-обещание. Он всегда приостанавливает выполнение в самом начале (строка 44) и конце (строка 48). Более того, метод

`get_return_object` (строка 36) создает возвращаемый объект, который возвращается вызывающей сопрограмме `createFuture` (строка 58). Более интересен класс `MyFuture`. Он содержит дескриптор `soho` (строка 12) для объекта-обещания. `MyFuture` использует дескриптор для управления объектом-обещанием. Он возобновляет выполнение объекта-обещания (строка 24), запрашивает у него результат (строка 25) и уничтожает его (строка 19). Возобновление сопрограммы необходимо, поскольку она никогда не выполняется автоматически (строка 44). Когда клиент вызывает `fut.get()` (строка 68) для получения результата, то объект-обещание неявно возобновляется (строка 24).

Вы можете попробовать выполнить эту программу в `Compiler Explorer`<sup>1</sup>.

```
promise_type::promise_type
promise_type::get_return_object
MyFuture::MyFuture
promise_type::initial_suspend
MyFuture::get
createFuture
promise_type::return_value
promise_type::final_suspend
res: 2021

MyFuture::~~MyFuture
promise_type::~~promise_type
```

Ленивый объект future

Что случится, если клиент не заинтересован в результате объекта `future`? Давайте попробуем.

Клиент не возобновляет сопрограмму

---

```
int main() {

    std::cout << '\n';

    auto fut = createFuture();
    // auto res = fut.get();
    // std::cout << "res: " << res << '\n';

    std::cout << '\n';

}
```

---

<sup>1</sup> <https://godbolt.org/z/EejWcj>.

Как можно было догадаться, объект-обещание никогда не выполняется, и методы `return_value` и `final_suspend` не выполняются.

```
int main() {
    std::cout << '\n';

    auto fut = createFuture();
    // auto res = fut.get();
    // std::cout << "res: " << res << '\n';

    std::cout << '\n';
}
```

Ленивый объект `future` не запускается



### Проблемы со временем жизни сопрограмм

Одной из проблем работы с сопрограммами является правильная обработка времени их жизни. В предыдущей программе `eagerFutureWithComments.cpp` я хранил сопрограмму в `std::shared_ptr`. Это критично, поскольку сопрограммы выполняются сразу. В программе `lazyFuture.cpp` вызов `final_suspend` всегда приостанавливает выполнение потока (строка 48): `std::suspend_always final_suspend()`. Соответственно, объект-обещание живет дольше клиента и `std::shared_ptr` больше не нужен. Возвращение `std::suspend_never` из метода `final_suspend` приведет в данном случае к неопределенному поведению (*undefined behavior*), поскольку клиент проживет дольше объекта-обещания. Поэтому время жизни результата заканчивается прежде, чем клиент об этом попросит.

Давайте опять изменим сопрограмму еще раз и запустим объект-обещание в отдельном потоке.

## 7.2.2 Выполнение на другом потоке

Сопрограмма полностью приостанавливается перед входом в сопрограмму `createFuture` (строка 67), поскольку метод `initial_suspend` возвращает `std::suspend_always` (строка 52). Поэтому объект-обещание может выполняться на другом потоке.

Выполнение объекта-обещания на другом потоке

```
1 // lazyFutureOnOtherThread.cpp
2
3 #include <coroutine>
4 #include <iostream>
5 #include <memory>
6 #include <thread>
7
8 template<typename T>
```

```
9  struct MyFuture {
10      struct promise_type;
11      using handle_type = std::coroutine_handle<promise_type>;
12      handle_type coro;
13
14      MyFuture(handle_type h): coro(h) {}
15      ~MyFuture() {
16          if ( coro ) coro.destroy();
17      }
18
19      T get() {
20          std::cout << "MyFuture::get: "
21                  << "std::this_thread::get_id(): "
22                  << std::this_thread::get_id() << '\n';
23
24          std::thread t([this] { coro.resume(); });
25          t.join();
26          return coro.promise().result;
27      }
28
29      struct promise_type {
30          promise_type(){
31              std::cout << "promise_type::promise_type: "
32                      << "std::this_thread::get_id(): "
33                      << std::this_thread::get_id() << '\n';
34          }
35          ~promise_type(){
36              std::cout << "promise_type::~~promise_type: "
37                      << "std::this_thread::get_id(): "
38                      << std::this_thread::get_id() << '\n';
39          }
40
41          T result;
42          auto get_return_object() {
43              return MyFuture{handle_type::from_promise(*this)};
44          }
45          void return_value(T v) {
46              std::cout << "promise_type::return_value: "
47                      << "std::this_thread::get_id(): "
```

---

```

48         << std::this_thread::get_id() << '\n';
49         std::cout << v << std::endl;
50         result = v;
51     }
52     std::suspend_always initial_suspend() {
53         return {};
54     }
55     std::suspend_always final_suspend() noexcept {
56         std::cout << "promise_type::final_suspend: "
57             << "std::this_thread::get_id(): "
58             << std::this_thread::get_id() << '\n';
59         return {};
60     }
61     void unhandled_exception() {
62         std::exit(1);
63     }
64 };
65 };
66
67 MyFuture<int> createFuture() {
68     co_return 2021;
69 }
70
71 int main() {
72
73     std::cout << '\n';
74
75     std::cout << "main: "
76         << "std::this_thread::get_id(): "
77         << std::this_thread::get_id() << '\n';
78
79     auto fut = createFuture();
80     auto res = fut.get();
81     std::cout << "res: " << res << '\n';
82
83     std::cout << '\n';
84
85 }

```

---

Я добавил несколько комментариев к этой программе, которые показывают идентификатор (id) выполняемого потока. Программа `lazyFutureOnOtherThread.cpp` похожа на предыдущую программу `lazyFuture.cpp`. Основное отличие – это метод `get` (строка 19). Вызов `std::thread t([this] { coro.resume(); });` (строка 24) возобновляет сопрограмму в другом потоке.

Вы можете попробовать выполнить эту программу на онлайн-компиляторе `Wandbox`<sup>1</sup>.

```
main: std::this_thread::get_id(): 139819561723776
      promise_type::promise_type: std::this_thread::get_id(): 139819561723776
      MyFuture::get: std::this_thread::get_id(): 139819561723776
      promise_type::return_value: std::this_thread::get_id(): 139819456755456
      promise_type::final_suspend: std::this_thread::get_id(): 139819456755456
res: 2021

      promise_type::~promise_type: std::this_thread::get_id(): 139819561723776
```

#### Выполнение на другом потоке

Следует сделать несколько замечаний о методе `get`. Крайне важно, чтобы объект-обещание, возобновляемый в отдельном потоке, завершался перед возвращением `coro.promise().result`.

Метод `get`, использующий `std::thread`

---

```
T get() {
    std::thread t([this] { coro.resume(); });
    t.join();
    return coro.promise().result;
}
```

---

Если бы я присоединил поток `t` после строки `return coro.promise().result`, у программы было бы неопределенное поведение. В следующей реализации метода `get` я использую `std::jthread`. Поскольку `std::jthread` автоматически присоединяется при выходе из области видимости, это слишком поздно.

Метод `get`, использующий `std::jthread`

---

```
T get() {
    std::jthread t([this] { coro.resume(); });
    return coro.promise().result;
}
```

---

В этом случае клиент, скорее всего, получит свой результат, прежде чем объект-обещание подготовит его при помощи метода `return_value`. Тогда `result` будет содержать произвольное значение, и его же получит `res`.

---

<sup>1</sup> <https://wandbox.org/permlink/JFVVj80Gxu6bnNkc>.

```

main: std::this_thread::get_id(): 139913381070720
      promise_type::promise_type: std::this_thread::get_id(): 139913381070720
      MyFuture::get: std::this_thread::get_id(): 139913381070720
      promise_type::return_value: std::this_thread::get_id(): 139913276102400
      promise_type::final_suspend: std::this_thread::get_id(): 139913276102400
res: -1

      promise_type::~~promise_type: std::this_thread::get_id(): 139913381070720

```

#### Выполнение на другом потоке

Есть другие способы, для того чтобы гарантировать, что выполнение потока будет завершено перед возвращением.

- Создать `std::jthread` в своей области видимости.

`std::jthread` в своей области видимости

---

```

T get() {
    {
        std::jthread t([this] { coro.resume(); });
    }
    return coro.promise().result;
}

```

---

- Сделать `std::jthread` временным объектом.

`std::jthread` как временный объект

---

```

T get() {
    std::jthread([this] { coro.resume(); });
    return coro.promise().result;
}

```

---

На самом деле мне не нравится последний вариант, поскольку может понадобиться несколько секунд, чтобы понять, что я просто вызвал конструктор `std::jthread`.

## 7.3 Модификация и обобщение генератора



Сиппи обрабатывает поток данных

Прежде чем я буду изменять и обобщать генератор для бесконечного потока данных, я хотел бы представить его как начальную точку для нашего путешествия. Я сознательно поместил в код много операций вывода и запрашиваю только три значения. Это упрощение и визуализация помогают понять ход выполнения программы.

Генератор, порождающий бесконечный поток данных

```

1  // infiniteDataStreamComments.cpp
2
3  #include <coroutine>
4  #include <memory>
5  #include <iostream>
6
7  template<typename T>
8  struct Generator {
9
10     struct promise_type;
11     using handle_type = std::coroutine_handle<promise_type>;
12
13     Generator(handle_type h): coro(h) {
14         std::cout << "Generator::Generator" << '\n';
15     }
16     handle_type coro;
17 
```



```

18     ~Generator() {
19         std::cout << "Generator::~~Generator" << '\n';
20         if ( coro ) coro.destroy();
21     }
22     Generator(const Generator&) = delete;
23     Generator& operator = (const Generator&) = delete;
24     Generator(Generator&& oth): coro(oth.coro) {
25         oth.coro = nullptr;
26     }
27     Generator& operator = (Generator&& oth) {
28         coro = oth.coro;
29         oth.coro = nullptr;
30         return *this;
31     }
32     int getNextValue() {
33         std::cout << "Generator::getNextValue" << '\n';
34         coro.resume();
35         return coro.promise().current_value;
36     }
37     struct promise_type {
38         promise_type() {
39             std::cout << "promise_type::promise_type" << '\n';
40         }
41
42         ~promise_type() {
43             std::cout << "promise_type::~~promise_type" << '\n';
44         }
45
46         std::suspend_always initial_suspend() {
47             std::cout << "promise_type::initial_suspend" << '\n'; \
48
49             return {};
50         }
51         std::suspend_always final_suspend() noexcept {
52             std::cout << "promise_type::final_suspend" << '\n';
53             return {};
54         }
55         auto get_return_object() {
56             std::cout << "promise_type::get_return_object" << '\n'; \
57

```

```
58         return Generator{handle_type::from_promise(*this)};
59     }
60
61     std::suspend_always yield_value(int value) {
62         std::cout << "promise_type::yield_value" << '\n'; \
63
64         current_value = value;
65         return {};
66     }
67     void return_void() {}
68     void unhandled_exception() {
69         std::exit(1);
70     }
71
72     T current_value;
73 };
74
75 };
76
77 Generator<int> getNext(int start = 10, int step = 10) {
78     std::cout << "getNext: start" << '\n';
79     auto value = start;
80     while (true) {
81         std::cout << "getNext: before co_yield" << '\n';
82         co_yield value;
83         std::cout << "getNext: after co_yield" << '\n';
84         value += step;
85     }
86 }
87
88 int main() {
89
90     auto gen = getNext();
91     for (int i = 0; i <= 2; ++i) {
92         auto val = gen.getNextValue();
93         std::cout << "main: " << val << '\n';
94     }
95
96 }
```

---

Выполнив эту программу в Compiler Explorer<sup>1</sup>, можно увидеть ход ее выполнения.

```

        promise_type::promise_type
        promise_type::get_return_object
    Generator::Generator
        promise_type::initial_suspend
    Generator::getNextValue
getNext: start
getNext: before co_yield
        promise_type::yield_value
main: 10
        Generator::getNextValue
getNext: after co_yield
getNext: before co_yield
        promise_type::yield_value
main: 20
        Generator::getNextValue
getNext: after co_yield
getNext: before co_yield
        promise_type::yield_value
main: 30
        Generator::~~Generator
        promise_type::~~promise_type

```

Генератор, создающий бесконечный поток данных

Давайте проанализируем ход выполнения.

Вызов `getNext()` (строка 87) приводит к созданию `Generator<int>`. Создается `promise_type` (строка 38), после чего следующий вызов `get_return_object` (строка 54) создает генератор (строка 56) и сохраняет его в локальной переменной. Результат этого вызова возвращается, когда сопрограмма приостанавливается в первый раз. Изначальная приостановка происходит немедленно (строка 48). Поскольку вызов метода `initial_suspend` возвращает ожидаемый объект `std::suspend_always` (строка 48), то выполнение продолжается до команды `co_yield value` (строка 79). Этот вызов отображается на вызов `yield_value(int value)` (строка 59), и текущее значение подготавливается в `current_value=value` (строка 61). Метод `yield_value(int value)` возвращает ожидаемый объект `std::suspend_always` (строка 59). Соответственно, выполнение сопрограммы приостанавливается, и управление возвращается функции `main`, после чего начинается цикл `for` (строка 89). Вызов `gen.getNextValue()` (строка 89) запускает выполнение сопрограммы через `co.resume()` (строка 34). Далее функция

<sup>1</sup> <https://godbolt.org/z/cTW9Gq>.

`getNextValue()` возвращает текущее значение, которое было подготовлено при помощи ранее вызванного метода `yield_value(int value)` (строка 59). Наконец, созданное значение выводится в строке 90, и цикл продолжается. В конце генератор и объект-обещание уничтожаются.

После детального анализа я хочу осуществить первое изменение хода выполнения.

### 7.3.1 Изменения

Фрагменты кода и номера строк основываются на ранее приведенной программе `infiniteDataStreamComments.cpp`. Я буду показывать только изменения.

#### 7.3.1.1 Сопрограмма не возобновляет выполнение

Когда я убираю возобновление сопрограммы (`gen.getNextValue()` в строке 89) и вывод значения (строка 90), то сопрограмма немедленно приостанавливается.

Не возобновляем сопрограмму

---

```
int main() {  
  
    auto gen = getNext();  
    for (int i = 0; i <= 2; ++i) {  
        // auto val = gen.getNextValue();  
        // std::cout << "main: " << val << '\n';  
    }  
  
}
```

---

Сопрограмма фактически никогда не выполняется. Поэтому генератор и объект-обещание просто создаются и уничтожаются.

---

```
int main() {  
  
    auto gen = getNext();  
    for (int i = 0; i <= 2; ++i) {  
        // auto val = gen.getNextValue();  
        // std::cout << "main: " << val << '\n';  
    }  
  
}
```

---

Не возобновляем сопрограмму

#### 7.3.1.2 `initial_suspend` никогда не приостанавливает сопрограмму

В нашей программе метод `initial_suspend` возвращает объект-ожидание `std::suspend_always` (строка 46). Как подсказывает имя, `std::suspend_always` всегда заставляет сопрограмму немедленно приостановить процесс выполнения. Давайте попробуем вернуть `std::suspend_never` вместо `std::suspend_always`.

---

`initial_suspend` никогда не приостанавливает выполнение сопрограммы

---

```
std::suspend_never initial_suspend() {
    std::cout << "promise_type::initial_suspend" << '\n';
    return {};
}
```

---

В этом случае сопрограмма немедленно выполняется и приостанавливается при вызове `yield_value` (строка 59). Последующий вызов `gen.getNextValue()` (строка 89) возобновляет сопрограмму и приводит к вызову метода `yield_value` еще раз. В результате начальное значение 10 игнорируется и сопрограмма возвращает значения 20, 30 и 40.

---

```
std::suspend_never initial_suspend() {
    std::cout << "promise_type::initial_suspend" << '\n';
    return {};
}
```

---

Не возобновляем сопрограмму

### 7.3.1.3 `yield_value` никогда не приостанавливает выполнение сопрограммы

Метод `yield_value` (строка 59) вызывается в результате вызова `co_yield value` и подготавливает `current_value` (строка 61). Функция возвращает `std::suspend_always` (строка 62) и поэтому приостанавливает выполнение сопрограммы. Соответственно, следующий вызов `gen.getNextValue()` (строка 89) должен возобновить выполнение сопрограммы. Изменим возвращаемое значение метода `yield_value` на `std::suspend_never`. Результат будет следующий:

`yield_value` никогда не приостанавливает выполнение сопрограммы

---

```
std::suspend_never yield_value(int value) {  
    std::cout << "promise_type::yield_value" << '\n';  
    current_value = value;  
    return {};  
}
```

---

Цикл `while` (строки 77-82) выполняется вечно, и сопрограмма ничего не возвращает.

---

```
std::suspend_never yield_value(int value) {  
    std::cout << "promise_type::yield_value" << '\n';  
    current_value = value;  
    return {};  
}
```

---

`yield_value` никогда не приостанавливает выполнение сопрограммы

Перестроить программу (генератор) `infiniteDataStreamComments.cpp` таким образом, чтобы она создавала конечное количество значений, довольно просто.

### 7.3.2 Обобщение

Вы могли удивиться, почему я никогда не использовал весь потенциал класса `Generator`. Давайте изменим нашу реализацию для получения последовательных элементов произвольного контейнера из стандартной библиотеки.

Генератор, успешно возвращающий каждый элемент

---

```

1  // coroutineGetElements.cpp
2
3  #include <coroutine>
4  #include <memory>
5  #include <iostream>
6  #include <string>
7  #include <vector>
8
9  template<typename T>
10 struct Generator {
11
12     struct promise_type;
13     using handle_type = std::coroutine_handle<promise_type>;
14
15     Generator(handle_type h): coro(h) {}
16
17     handle_type coro;
18
19     ~Generator() {
20         if ( coro ) coro.destroy();
21     }
22     Generator(const Generator&) = delete;
23     Generator& operator = (const Generator&) = delete;
24     Generator(Generator&& oth): coro(oth.coro) {
25         oth.coro = nullptr;
26     }
27     Generator& operator = (Generator&& oth) {
28         coro = oth.coro;
29         oth.coro = nullptr;
30         return *this;
31     }

```

```
32     T getNextValue() {
33         coro.resume();
34         return coro.promise().current_value;
35     }
36     struct promise_type {
37         promise_type() {}
38
39         ~promise_type() {}
40
41         std::suspend_always initial_suspend() {
42             return {};
43         }
44         std::suspend_always final_suspend() noexcept {
45             return {};
46         }
47         auto get_return_object() {
48             return Generator{handle_type::from_promise(*this)};
49         }
50
51         std::suspend_always yield_value(const T value) {
52             current_value = value;
53             return {};
54         }
55         void return_void() {}
56         void unhandled_exception() {
57             std::exit(1);
58         }
59
60         T current_value;
61     };
62
63 };
64
65 template <typename Cont>
66 Generator<typename Cont::value_type> getNext(Cont cont) {
67     for (auto c: cont) co_yield c;
68 }
69
```



```
70 int main() {
71
72     std::cout << '\n';
73
74     std::string helloWorld = "Hello world";
75     auto gen = getNext(helloWorld);
76     for (int i = 0; i < helloWorld.size(); ++i) {
77         std::cout << gen.getNextValue() << " ";
78     }
79
80     std::cout << "\n\n";
81
82     auto gen2 = getNext(helloWorld);
83     for (int i = 0; i < 5 ; ++i) {
84         std::cout << gen2.getNextValue() << " ";
85     }
86
87     std::cout << "\n\n";
88
89     std::vector myVec{1, 2, 3, 4 ,5};
90     auto gen3 = getNext(myVec);
91     for (int i = 0; i < myVec.size() ; ++i) {
92         std::cout << gen3.getNextValue() << " ";
93     }
94
95     std::cout << '\n';
96
97 }
```

В этом примере генератор создается и используется три раза. В первых двух случаях `gen` (строка 76) и `gen2` (строка 83) инициализируются при помощи `std::string helloWorld`, в то время как `gen3` использует `std::vector<int>` (строка 91). Вывод программы вряд ли вас удивит. Строка 78 успешно возвращает все символы строки `helloWorld`, строка 85 – только первые 5 символов, а строка 93 возвращает элементы `std::vector<int>`.

Запустите эту программу с помощью Compiler Explorer<sup>1</sup>.

---

<sup>1</sup> <https://godbolt.org/z/j9znva>.

```
1 // coroutineDefinition.cpp
2
3 #include <coroutine>
4 #include <memory>
5 #include <iostream>
6 #include <string>
7 #include <vector>
8
9 template<typename T>
10 struct Generator {
11
12     struct promise_type;
13     using handle_type = std::coroutine_handle<promise_type>;
14     Generator(handle_type h): coro(h) {}
15
16     handle_type coro;
17
18 }
```

Генератор, успешно возвращающий каждый элемент

Подводя итоги: реализация `Generator<T>` почти идентична предыдущему примеру. Принципиальным отличием является сопрограмма `getNext`.

`getNext`

---

```
template <typename Cont>
Generator<typename Cont::value_type> getNext(Cont cont) {
    for (auto c: cont) co_yield c;
}
```

---

Шаблонная функция `getNext` принимает в качестве своего аргумента контейнер и перебирает все элементы этого контейнера. После каждой итерации происходит приостановка сопрограммы. Возвращаемый тип `Generator<typename Cont::value_type>` может показаться вам странным. Здесь `Cont::value_type` – это зависимый шаблонный параметр, для которого компилятору нужна подсказка. По умолчанию компилятор предполагает нетиповой параметр, если он может это интерпретировать как тип или «не тип». Именно поэтому мне нужно было поместить `typename` перед `Cont::value_type`.

## 7.4 Различные потоковые архитектуры, основанные на задачах



Сиппи копает в саду

Прежде чем я буду модифицировать пример из раздела по `co_await`, я хотел бы сделать пример с ожидающим (`awaiter`) потоком задач более прозрачным.

### 7.4.1 Прозрачная архитектура ожидающего потока задач

Я просто добавил вывод нескольких сообщений к программе `startJob.cpp`.

Запуск задач по запросу

---

```

1  // startJobWithComments.cpp
2
3  #include <coroutine>
4  #include <iostream>
5
6  struct MySuspendAlways {
7      bool await_ready() const noexcept {
8          std::cout << " MySuspendAlways::await_ready" << '\n';
9          return false;
10     }
11     void await_suspend(std::coroutine_handle<>) const noexcept {
12         std::cout << " MySuspendAlways::await_suspend" << '\n';
13     }
14 }
```

```
15     void await_resume() const noexcept {
16         std::cout << " MySuspendAlways::await_resume" << '\n';
17     }
18 };
19
20 struct MySuspendNever {
21     bool await_ready() const noexcept {
22         std::cout << " MySuspendNever::await_ready" << '\n';
23         return true;
24     }
25     void await_suspend(std::coroutine_handle<>) const noexcept {
26         std::cout << " MySuspendNever::await_suspend" << '\n';
27     }
28 }
29 void await_resume() const noexcept {
30     std::cout << " MySuspendNever::await_resume" << '\n';
31 }
32 };
33
34 struct Job {
35     struct promise_type;
36     using handle_type = std::coroutine_handle<promise_type>;
37     handle_type coro;
38     Job(handle_type h): coro(h){}
39     ~Job() {
40         if ( coro ) coro.destroy();
41     }
42     void start() {
43         coro.resume();
44     }
45
46
47     struct promise_type {
48         auto get_return_object() {
49             return Job{handle_type::from_promise(*this)};
50         }
51         MySuspendAlways initial_suspend() {
52             std::cout << " Job prepared" << '\n';
```

---

```

53         return {};
54     }
55     MySuspendAlways final_suspend() noexcept {
56         std::cout << "Job finished" << '\n';
57         return {};
58     }
59     void return_void() {}
60     void unhandled_exception() {}
61
62 };
63 };
64
65 Job prepareJob() {
66     co_await MySuspendNever();
67 }
68
69 int main() {
70
71     std::cout << "Before job" << '\n';
72
73     auto job = prepareJob();
74     job.start();
75
76     std::cout << "After job" << '\n';
77
78 }

```

---

В этой программе были заменены предопределенные объекты `std::suspend_always` и `std::suspend_never` на свои ожидаемые объекты `MySuspendAlways` (строка 6) и `MySuspendNever` (строка 20). Я использую их в строках 51, 55 и 66. Эти ожидаемые объекты имитируют поведение предопределенных ожидаемых объектов, но вдобавок выводят сообщения. Благодаря использованию `std::cout` методы `await_ready`, `await_suspend` и `await_resume` не могут быть объявлены как `constexpr`.

Скриншот выполнения программы хорошо показывает, как это происходит. Вы можете сами попробовать запустить приведенный пример прямо в `Compiler Explorer`<sup>1</sup>.

---

<sup>1</sup> <https://godbolt.org/z/T5rcE4>.

```

Before job
    Job prepared
        MySuspendAlways::await_ready
        MySuspendAlways::await_suspend
        MySuspendAlways::await_resume
        MySuspendNever::await_ready
        MySuspendNever::await_resume
    Job finished
        MySuspendAlways::await_ready
        MySuspendAlways::await_suspend
After job

```

Запуск задач по запросу

Функция `initial_suspend` (строка 51) выполняется в начале сопрограммы, функция `final_suspend` – в ее конце (строка 55). Вызов `prepareJob()` (строка 73) запускает создание объекта-сопрограммы, вызов `job.start()` приводит к ее возобновлению и завершению (строка 74). Поэтому методы `await_ready`, `await_suspend` и `await_resume` класса `MySuspendAlways` выполняются. Когда вы не возобновляете ожидаемый объект, как в случае с объектом, возвращенным методом `final_suspend`, функция `await_resume` не выполняется. Для сравнения, метод класса `MySuspendNever` уже готов, поскольку `await_ready` возвращает `true` и поэтому не приводит к приостановке.

Благодаря комментариям у вас должно сложиться хотя бы элементарное понимание того, как работает процесс ожидания. Теперь самое время его немного изменить.

## 7.4.2 Автоматическое возобновление ожидающей задачи

В предыдущем примере я явным образом запустил задачу.

Явный запуск задачи

```

Before job
    Job prepared
        MySuspendAlways::await_re
        MySuspendAlways::await_su
        MySuspendAlways::await_re
        MySuspendNever::await rea

```

Явный вызов `job.start()` был необходим, поскольку метод `await_ready` в `MySuspendAlways` всегда возвращает `false`. Теперь давайте допустим, что `await_ready` может вернуть `true` или `false` и задача не запускается явно. Краткое напоминание: когда `await_ready` возвращает `true`, то функция `await_resume` непосредственно вызывается, а `await_suspend` – нет.

Автоматическое возобновление ожидающей задачи

---

```

1  // startJobWithAutomaticResumption.cpp
2
3  #include <coroutine>
4  #include <functional>
5  #include <iostream>
6  #include <random>
7
8  std::random_device seed;
9  auto gen = std::bind_front(std::uniform_int_distribution<>(0,1),
10                             std::default_random_engine(seed()));
11
12  struct MySuspendAlways {
13      bool await_ready() const noexcept {
14          std::cout << " MySuspendAlways::await_ready" << '\n';
15          return gen();
16      }
17      bool await_suspend(std::coroutine_handle<> handle) const noexcept {
18          std::cout << " MySuspendAlways::await_suspend" << '\n';
19          handle.resume();
20          return true;
21      }
22  }
23  void await_resume() const noexcept {
24      std::cout << " MySuspendAlways::await_resume" << '\n';
25  }
26  };
27
28  struct Job {
29      struct promise_type;
30      using handle_type = std::coroutine_handle<promise_type>;
31      handle_type coro;
32      Job(handle_type h): coro(h){}
33      ~Job() {
34          if ( coro ) coro.destroy();
35      }
36

```

```
37     struct promise_type {
38         auto get_return_object() {
39             return Job{handle_type::from_promise(*this)};
40         }
41         MySuspendAlways initial_suspend() {
42             std::cout << "Job prepared" << '\n';
43             return {};
44         }
45         std::suspend_always final_suspend() noexcept {
46             std::cout << "Job finished" << '\n';
47             return {};
48         }
49         void return_void() {}
50         void unhandled_exception() {}
51     };
52 };
53 };
54
55 Job performJob() {
56     co_await std::suspend_never();
57 }
58
59 int main() {
60
61     std::cout << "Before jobs" << '\n';
62
63     performJob();
64     performJob();
65     performJob();
66     performJob();
67
68     std::cout << "After jobs" << '\n';
69
70 }
```

---

Теперь сопрограмма называется `performJob` и запускается автоматически. `gen` (строка 9) – это генератор случайных чисел, выдающий 0 или 1. Он использует для своей работы стандартный генератор случайных чисел, инициализируя его начальным значением. Благодаря `std::bind_front` я могу соединить его с `std::uniform_int_distribution` для получения вызываемого объекта, который при вызове будет выдавать 0 или 1.

Я убрал из этого примера стандартные ожидаемые объекты в C++, за исключением ожидаемого объекта `MySuspendAlways`, который является возвращаемым



типом при вызове метода `initial_suspend` (строка 41). Метод `await_ready` (строка 13) возвращает логическое значение. Когда оно равно `true`, управление непосредственно переходит к методу `await_resume` (строка 23), а когда оно равно `false`, то сопрограмма немедленно приостанавливается, а функция `await_suspend` (строка 17) выполняется. Функция `await_suspend` получает дескриптор сопрограммы и использует его для возобновления сопрограммы (строка 19). Вместо возвращения значения `true` функция `await_suspend` может также вернуть `void`.

Следующий скриншот показывает, что когда `await_ready` возвращает `true`, то вызывается `await_resume`, а когда `await_ready` возвращает `false`, то вызывается функция `await_suspend`.

Проверьте эту программу в Compiler Explorer<sup>1</sup>.

```
Before jobs
  Job prepared
    MySuspendAlways::await_ready
    MySuspendAlways::await_suspend
    MySuspendAlways::await_resume
  Job finished
  Job prepared
    MySuspendAlways::await_ready
    MySuspendAlways::await_resume
  Job finished
  Job prepared
    MySuspendAlways::await_ready
    MySuspendAlways::await_resume
  Job finished
  Job prepared
    MySuspendAlways::await_ready
    MySuspendAlways::await_resume
  Job finished
After jobs
```

Автоматическое возобновление ожидающего объекта

Давайте улучшим представленную программу еще больше и возобновим ожидающий объект на отдельном потоке.

### 7.4.3 Автоматическое возобновление ожидающего объекта на отдельном потоке

Следующая программа основана на предыдущем примере.

<sup>1</sup> <https://godbolt.org/z/8b1Y14>.

Автоматическое возобновление ожидающего объекта на отдельном потоке

---

```
1  // startJobWithAutomaticResumptionOnThread.cpp
2
3  #include <coroutine>
4  #include <functional>
5  #include <iostream>
6  #include <random>
7  #include <thread>
8  #include <vector>
9
10 std::random_device seed;
11 auto gen = std::bind_front(std::uniform_int_distribution<>(0,1),
12                             std::default_random_engine(seed()));
13
14 struct MyAwaitable {
15     std::jthread& outerThread;
16     bool await_ready() const noexcept {
17         auto res = gen();
18         if (res) std::cout << " (executed)" << '\n';
19         else std::cout << " (suspended)" << '\n';
20         return res;
21     }
22     void await_suspend(std::coroutine_handle<> h) {
23         outerThread = std::jthread([h] { h.resume(); });
24     }
25     void await_resume() {}
26 };
27
28
29 struct Job{
30     static inline int JobCounter{1};
31     Job() {
32         ++JobCounter;
33     }
34
35     struct promise_type {
36         int JobNumber{JobCounter};
37         Job get_return_object() { return {}; }
38         std::suspend_never initial_suspend() {
```

---

```

39         std::cout << "Job " << JobNumber << "prepared on thread"
40             << std::this_thread::get_id();
41         return {};
42     }
43     std::suspend_never final_suspend() noexcept {
44         std::cout << "Job " << JobNumber << "finished on thread"
45             << std::this_thread::get_id() << '\n';
46         return {};
47     }
48     void return_void() {}
49     void unhandled_exception() { }
50 };
51 };
52
53 Job performJob(std::jthread& out) {
54     co_await MyAwaitable{out};
55 }
56
57 int main() {
58
59     std::vector<std::jthread> threads(8);
60     for (auto& thr: threads) performJob(thr);
61
62 }

```

---

Главным отличием от предыдущей программы является новый ожидаемый объект `MyAwaitable`, используемый в сопрограмме `performJob` (строка 54). Сам объект-сопрограмма, возвращаемый `performJob`, прост. По сути, его методы `initial_suspend` (строка 38) и `final_suspend` (строка 43) возвращают `std::suspend_never`. Кроме того, оба этих метода показывают `JobNumber` выполняемого задания и ID потока, в котором происходит выполнение. Скриншот показывает, какая сопрограмма выполняется сразу, а какая приостанавливается. Благодаря выводу идентификатора потока можно увидеть, что приостановленные сопрограммы возобновляются на отдельных потоках.

Запустите эту программу на Wandbox<sup>1</sup>.

---

<sup>1</sup> <https://wandbox.org/permlink/skHgWKFO5YAw8Dm>.

```
Job 1 prepared on thread 140434982274944 (executed)
Job 1 finished on thread 140434982274944
Job 2 prepared on thread 140434982274944 (suspended)
Job 3 prepared on thread 140434982274944 (suspended)
Job 4 prepared on thread 140434982274944 (suspended)
Job 2 finished on thread 140434877310720
Job 5 prepared on thread 140434982274944 (executed)
Job 5 finished on thread 140434982274944
Job 6 prepared on thread 140434982274944 (suspended)
Job 7 prepared on thread 140434982274944 (suspended)
Job 3 finished on thread 140434868918016
Job 8 prepared on thread 140434982274944 (executed)
Job 8 finished on thread 140434982274944
Job 4 finished on thread 140434860525312
Job 6 finished on thread 140434852132608
Job 7 finished on thread 140434843739904
```

Автоматическое возобновление ожидающего объекта в отдельном потоке

Давайте обсудим интересное поведение потока управления данной программы. Строка 59 создает 8 потоков при помощи конструктора по умолчанию, к которым сопрограмма `performJob` (строка 53) обращается по ссылке. Далее эта ссылка становится аргументом для создания `MyAwaitable` (строка 54). В зависимости от значения `res` (строка 17) и значения, возвращенного `await_ready`, ожидаемый объект возобновляет выполнение (`res` равно `true`) или приостанавливается (`res` равно `false`). В случае, когда `MyAwaitable` приостанавливается, выполняется функция `await_suspend` (строка 22). Благодаря присваиванию `outerThread` (строка 23) он становится работающим потоком. Выполняемые потоки должны прожить дольше, чем время жизни сопрограммы. Именно по этой причине у всех потоков область видимости – это функция `main`.



### Краткая информация

- ♦ Когда вы хотите синхронизировать потоки более одного раза, то у вас есть много вариантов. Вы можете использовать условные переменные, `std::atomic_flag`, `std::atomic<bool>` или семафоры. В этом разделе дается ответ на вопрос: какой из этих вариантов самый быстрый? Полученные результаты замеров показывают, что вариант с условными переменными самый медленный, а атомарные флаги – это самый быстрый вариант для синхронизации потоков. Быстродействие `std::atomic<bool>` является средним. Семафоры практически так же быстры, как и атомарные флаги.
- ♦ В разделе по сопрограммам была показана жадная (*greedy*) сопрограмма, использующая `co_return`. Объект `future` – это идеальная начальная точка, для того чтобы сделать сопрограмму отложенной (*lazy*) и позволить ей выполняться в отдельном потоке.

- ♦ Изменения генератора бесконечного потока данных показывают его природу. Когда метод `initial_suspend` возвращает `std::suspend_never`, то сопрограмма начинает выполняться сразу и игнорирует первое значение. В отличие от этого, возвращение `std::suspend_never` из метода `yield_value` приводит к бесконечному циклу. Если вы забываете возобновить сопрограмму, то она просто никогда не выполнится.
- ♦ Генератор `Generator<T>` применим для ряда случаев. Вместо бесконечного потока данных он может успешно возвращать элементы произвольного контейнера из стандартной библиотеки шаблонов.
- ♦ Реализация своих ожидаемых объектов `MySuspendNever` и `MySuspendAlways` делает работу с ожидаемыми объектами прозрачной. Адаптация `MySuspendAlways` позволяет создать ожидаемый объект, который при необходимости сам себя возобновляет.
- ♦ Изменения ожидаемого объекта дают возможность автоматически возобновлять выполнение сопрограммы на отдельном потоке.

# Эпилог

Поздравляю! Если вы читаете эти строки, то вы освоили непростой и захватывающий стандарт C++20. C++20 – это стандарт, который, скорее всего, будет иметь такое же значение для C++, как и два других наиболее значимых стандарта: C++98 и C++11. В связи со стандартом C++11 в сообществе C++ используются следующие термины:

- наследие C++ (Legacy) : C++ 98 и C++03;
- современный C++: C++11, C++14 и C++17;
- <пока не ясно> C++20.

Я не знаю, какое слово будет использовано для C++20 в будущем. Я уверен только в том, что C++20 начинает новую эру в C++. Позвольте мне напомнить, почему *Большая четверка* (Big Four) изменила то, как мы программируем на C++.

- **Концепты**: концепты переворачивают способ, которым мы думаем и пишем обобщенный код. Благодаря им мы впервые можем рассуждать о программе в семантических категориях, таких как Number или Ordering.
- **Модули**: модули – это стартовая точка программных компонент. Модули помогают преодолеть недостатки традиционных заголовочных файлов и макросов.
- **Диапазоны**: библиотека диапазонов расширяет стандартную библиотеку шаблонов при помощи функциональных идей. Алгоритмы могут работать непосредственно с контейнерами, могут выполняться отложено и могут совмещаться друг с другом.
- **Сопрограммы**: благодаря сопрограммам асинхронное программирование становится первоклассным гражданином в C++. Сопрограммы преобразуют блокирующие вызовы функций в ожидание и крайне полезны в системах, основанных на обработке событий, таких как симуляции, серверы или пользовательские интерфейсы.

C++20 – это начало. Много чего еще нужно сделать в C++23 для полной интеграции и использования потенциала *Большой четверки* в C++. Позвольте мне дать вам несколько идей насчет ближайшего будущего C++.

- Стандартную библиотеку шаблонов разработал Александр Степанов<sup>1</sup>, придерживаясь понятия концептов. Но при этом интеграция концептов все еще отсутствует в C++20.

---

<sup>1</sup> [https://en.wikipedia.org/wiki/Alexander\\_Stepanov](https://en.wikipedia.org/wiki/Alexander_Stepanov).

- Мы можем ожидать модульной стандартной библиотеки шаблонов и надеяться на систему пакетов в C++.
- Многие алгоритмы, известные из функционального программирования, все еще отсутствуют в библиотеке диапазонов. Будущий стандарт C++ должен улучшить взаимодействие алгоритмов над диапазонами со стандартными контейнерами.
- У нас еще нет сопрограмм. У нас есть только библиотека для построения мощных сопрограмм. Библиотека сопрограмм с большой вероятностью появится в C++23.

В главе «C++23 и не только» я дам больше деталей о ближайшем будущем C++. Говоря короче, у C++ яркое будущее.

*Rainer Grimm*

# **ДОПОЛНИТЕЛЬНАЯ ИНФОРМАЦИЯ**



## 8. C++23 и не только

Бытует мнение, что за «большим» обновлением стандарта C++ следует небольшое. Но в случае с C++23 это, вероятно, будет не так. Ожидается, что стандарт C++23 принесет настолько же значительные улучшения, как и C++20. Виль Вoutilейнен (Ville Voutilainen) в P0592R4<sup>1</sup> предложил «явно обозначить общий план для C++23», дающий общее представление о будущем стандарта C++23. Виль называет семь основных нововведений, которые ожидаются в новом стандарте:

- C++23:
  - ◆ поддержка сопрограмм на уровне библиотеки;
  - ◆ модульная стандартная библиотека;
  - ◆ исполнители (executors);
  - ◆ библиотека для работы с сетевыми приложениями (networking);
- C++23 или позже:
  - ◆ рефлексия (reflection);
  - ◆ сопоставление с образцом (pattern matching);
  - ◆ контракты.

Первые четыре пункта планируется ввести с выходом C++23, а у оставшихся нет какого-то определенного срока выхода. Весьма вероятно, что рефлексия, сопоставление с образцом и контракты тоже будут скоро добавлены в стандарт C++.

«Предсказание является очень сложным, особенно если речь идет о будущем» (Нильс Бор<sup>2</sup>). Поэтому вам следует воспринимать эту главу как мою попытку предсказать будущее C++.

---

<sup>1</sup> <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0592r4.html>.

<sup>2</sup> <https://www.goodreads.com/quotes/23796-prediction-is-very-difficult-especially-about-the-future>.

## 8.1 C++23

Библиотека сопрограмм, модульная стандартная библиотека и исполнители имеют нечто общее: они рассматриваются как часть C++23.

### 8.1.1 Библиотека сопрограмм

Сопрограммы в C++20 – это не более чем фреймворк (framework) для реализации конкретных сопрограмм. Это значит, что реализация сопрограмм – это ответственность разработчика. Библиотека `srpcoro`<sup>1</sup> от Льюиса Бейкера (Lewis Baker) дает некоторое представление о том, как библиотека сопрограмм может выглядеть. Его библиотека предоставляет то, чего нет у C++20: высокоуровневые сопрограммы.



#### Использование `srpcoro`

Библиотека `srpcoro` основана на технической спецификации по сопрограммам. Термин «техническая спецификация» является предварительной версией функциональности сопрограмм, которую мы получили в C++20. Вероятно, что Льюис в будущем выполнит портирование библиотеки `srpcoro` от технической спецификации к сопрограммам, как они определены в C++20. Эта библиотека может использоваться в Windows (Visual Studio 2017) или Linux (Clang 5.0/6.0 и `libc++`). Для своих экспериментов я использовал следующую командную строку:

Командная строка для `srpcoro`

---

```
clang++ -std=c++17 -fcoroutines-ts -Iinclude -stdlib=libc++ libcppcoro.a
  cppcoroTask.cpp -pthread
```

---

- ♦ `-std=c++17` – поддержка версии C++17;
- ♦ `-fcoroutines-ts` – поддержка сопрограмм из технической спецификации;
- ♦ `-Iinclude` – заголовочные файлы для `srpcoro`;
- ♦ `-stdlib=libc++` – реализация стандартной библиотеки от LLVM<sup>2</sup>;
- ♦ `libcppcoro.a` – библиотека `srpcoro`.

Как уже упоминалось, когда `srpcoro` будет основываться на сопрограммах из C++20, вы сможете использовать их с каждым компилятором, поддерживающим C++20. Также вы получите некоторое впечатление о том, как конкретные сопрограммы будут выглядеть в C++23.

В оставшейся части раздела по библиотеке по сопрограммам я хотел бы показать несколько примеров, демонстрирующих всю мощь сопрограмм. Моя демонстрация начнется с типов сопрограмм.

<sup>1</sup> <https://github.com/lewisbaker/cppcoro>.

<sup>2</sup> <https://en.wikipedia.org/wiki/LLVM>.

### 8.1.1.1 Типы сопрограмм

В `cppcoro` есть несколько типов задач (tasks) и генераторов (generators).

#### 8.1.1.1.1 `task<T>`

Что такое задача (task)? Ниже приводится определение, используемое в `cppcoro`:

- задача представляет собой асинхронное вычисление, которое выполняется отложено (лениво, lazily) в том смысле, что выполнение сопрограммы не начинается до того момента, пока кто-то не затребует (await) ее выполнение.

Задача – это сопрограмма. В следующем примере функция `main` ожидает функцию `first`. Функция `first` ожидает функции `second`. А функция `second` ожидает функции `third`.

Спящие сопрограммы

---

```

1  // cppcoroTask.cpp
2
3  #include <chrono>
4  #include <iostream>
5  #include <string>
6  #include <thread>
7
8  #include <cppcoro/sync_wait.hpp>
9  #include <cppcoro/task.hpp>
10
11 using std::chrono::high_resolution_clock;
12 using std::chrono::time_point;
13 using std::chrono::duration;
14
15 using namespace std::chrono_literals;
16
17 auto getTimeSince(const time_point<high_resolution_clock>& start) {
18
19     auto end = high_resolution_clock::now();
20     duration<double> elapsed = end - start;
21     return elapsed.count();
22
23 }
24
25 cppcoro::task<> third(const time_point<high_resolution_clock>& start) {
26
27     std::this_thread::sleep_for(1s);
28     std::cout << "Third waited " << getTimeSince(start) << " seconds." << '\n';
29
30     co_return;
31
32 }
```

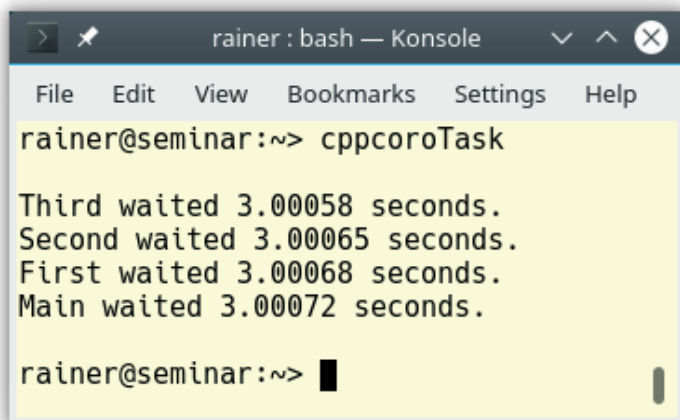
```
33
34 cppcoro::task<> second(const time_point<high_resolution_clock>& start) {
35
36     auto thi = third(start);
37     std::this_thread::sleep_for(1s);
38     co_await thi;
39
40     std::cout << "Second waited " << getTimeSince(start) << " seconds." << '\n';
41
42 }
43
44 cppcoro::task<> first(const time_point<high_resolution_clock>& start) {
45
46     auto sec = second(start);
47     std::this_thread::sleep_for(1s);
48     co_await sec;
49
50     std::cout << "First waited " << getTimeSince(start) << " seconds." << '\n';
51
52 }
53
54 int main() {
55
56     std::cout << '\n';
57
58     auto start = high_resolution_clock::now();
59     cppcoro::sync_wait(first(start));
60
61     std::cout << "Main waited " << getTimeSince(start) << " seconds." << '\n';
62
63     std::cout << '\n';
64
65 }
```

---

Признаю, что эта программа не делает ничего полезного, но она помогает понять работу сопрограмм.

Функция `main` не может быть сопрограммой. `cppcoro::sync_wait` (строка 59) часто выполняет роль, как и в этом случае, стартовой задачи высшего уровня и ожидает, пока задача не завершится. Сопрограмма `first`, похожая на остальные сопрограммы, получает в качестве аргумента время начала и выводит время выполнения. Что делает сопрограмма `first`? Она запускает сопрограмму `second` (строки 36 и 46), которая немедленно приостанавливается, спит в течение секунды и затем возобновляет сопрограмму при помощи ее дескриптора `sec` (строки 38 и 48). Сопрограмма `second` выполняет то же самое. Сопрограмма `third` – это сопрограмма, которая ничего не возвращает и не ждет другой со-

программы. Когда `third` заканчивает свое выполнение, то выполняются остальные сопрограммы. Соответственно, каждая сопрограмма занимает 3 секунды.



```
rainer : bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~> cppcoroTask

Third waited 3.00058 seconds.
Second waited 3.00065 seconds.
First waited 3.00068 seconds.
Main waited 3.00072 seconds.

rainer@seminar:~> █
```

Спящие сопрограммы

Давайте сейчас изменим программу. Что произойдет, если сопрограмма заснет сразу после вызова `co_await`?

Сразу засыпающие сопрограммы

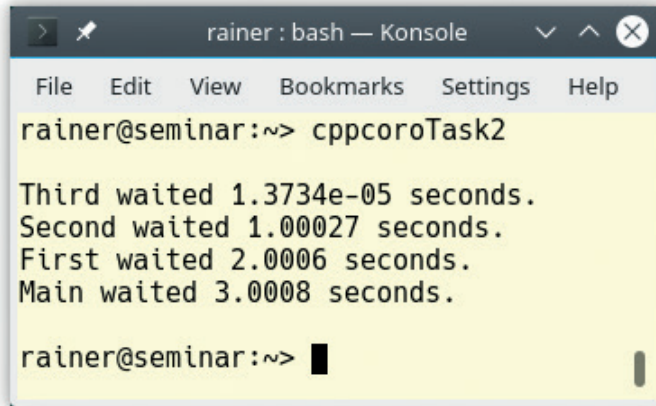
---

```
1 // cppcoroTask2.cpp
2
3 #include <chrono>
4 #include <iostream>
5 #include <string>
6 #include <thread>
7
8 #include <cppcoro/sync_wait.hpp>
9 #include <cppcoro/task.hpp>
10
11 using std::chrono::high_resolution_clock;
12 using std::chrono::time_point;
13 using std::chrono::duration;
14
15 using namespace std::chrono_literals;
16
17 auto getTimeSince(const time_point<::high_resolution_clock>& start) {
18
19     auto end = high_resolution_clock::now();
20     duration<double> elapsed = end - start;
21     return elapsed.count();
22
23 }
```

```
24
25 cppcoro::task<> third(const time_point<high_resolution_clock>& start) {
26
27     std::cout << "Third waited " << getTimeSince(start) << " seconds." << '\n';
28     std::this_thread::sleep_for(1s);
29     co_return;
30
31 }
32
33 cppcoro::task<> second(const time_point<high_resolution_clock>& start) {
34
35     auto thi = third(start);
36     co_await thi;
37
38     std::cout << "Second waited " << getTimeSince(start) << " seconds." << '\n';
39     std::this_thread::sleep_for(1s);
40
41 }
42
43 cppcoro::task<> first(const time_point<high_resolution_clock>& start) {
44
45     auto sec = second(start);
46     co_await sec;
47
48     std::cout << "First waited " << getTimeSince(start) << " seconds." << '\n';
49     std::this_thread::sleep_for(1s);
50
51 }
52
53 int main() {
54
55     std::cout << '\n';
56
57     auto start = ::high_resolution_clock::now();
58
59     cppcoro::sync_wait(first(start));
60
61     std::cout << "Main waited " << getTimeSince(start) << " seconds." << '\n';
62
63     std::cout << '\n';
64
65 }
```

---

Результат следующий: функция `main` ждет 3 секунды, но каждая из по очереди вызываемых сопрограмм ждет на одну секунду меньше.



```
rainer : bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~> cppcoroTask2

Third waited 1.3734e-05 seconds.
Second waited 1.00027 seconds.
First waited 2.0006 seconds.
Main waited 3.0008 seconds.

rainer@seminar:~> █
```

Ждущие сопрограммы

Следующая сопрограмма, которую предоставляет `cppcoro`, – это `Generator<T>`.

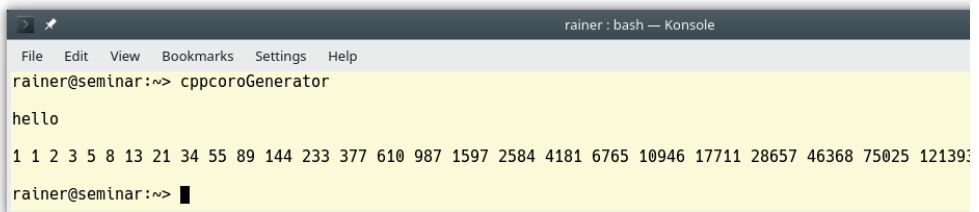
#### 8.1.1.1.2 `Generator<T>`

Ниже приводится определение генератора из `cppcoro`:

- генератор представляет собой тип сопрограммы, которая производит последовательность значений типа `T`, где сами значения производятся отложено и синхронно.

Рассмотрим программу `cppcoroGenerator.cpp`, которая показывает работу двух генераторов.

Use of two generators



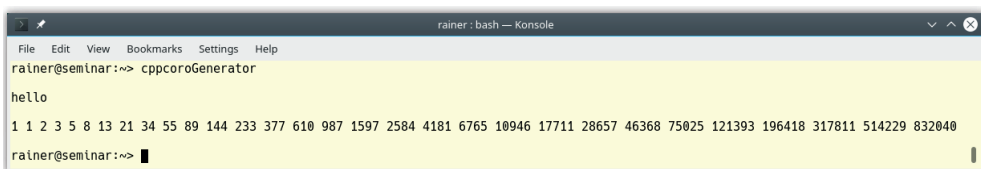
```
rainer : bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~> cppcoroGenerator

hello
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 12139:

rainer@seminar:~> █
```

Первая сопрограмма `hello` по запросу возвращает следующий символ, а сопрограмма `fibonacci` возвращает следующее число Фибоначчи. Сопрограмма `fibonacci` создает бесконечный поток данных.

Разберемся с тем, что происходит в строке 33. Цикл приводит к выполнению сопрограммы. Первая итерация запускает сопрограммы, возвращая значение через `co_yield b` (строка 18), и приостанавливается. Последующие вызовы из цикла `for` возобновляют сопрограмму `fibonacci` и возвращают следующее число Фибоначчи.



```
rainier : bash — Konsole
File Edit View Bookmarks Settings Help
rainier@semlnar:~> cppcoroGenerator
hello
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040
rainier@semlnar:~> █
```

Выполнение двух генераторов

`cppcoro` также предоставляет еще больше ожидаемых типов (`awaitable types`).



### 8.1.1.2 Ожидаемые типы

Библиотека `cppcoro` поддерживает следующие ожидаемые типы:

- `single_consumer_event`
- `single_consumer_async_auto_reset_event`
- `async_mutex`
- `async_manual_reset_event`
- `async_auto_reset_event`
- `async_latch`
- `sequence_barrier`
- `multi_producer_sequencer`
- `single_producer_sequencer`

Рассмотрим типы `single_consume_event` и `async_mutex`.

#### 8.1.1.2.1 `single_consume_event`

`single_consume_event`, согласно документации, – это простой тип события с ручным сбросом, который поддерживает только одну ожидающую его сопрограмму. Класс `single_consume_event` предоставляет новый способ для однократной синхронизации потоков.

Однократная синхронизация потоков через `cppcoro`

---

```

1  // cppcoroProducerConsumer.cpp
2
3  #include <cppcoro/single_consumer_event.hpp>
4  #include <cppcoro/sync_wait.hpp>
5  #include <cppcoro/task.hpp>
6
7  #include <future>
8  #include <iostream>
9  #include <string>
10 #include <thread>
11 #include <chrono>
12
13 cppcoro::single_consumer_event event;
14
15 cppcoro::task<> consumer() {
16
17     auto start = std::chrono::high_resolution_clock::now();
18
19     co_await event; // suspended until some thread calls event.set()
20
21     auto end = std::chrono::high_resolution_clock::now();
22     std::chrono::duration<double> elapsed = end - start;
23     std::cout << "Consumer waited " << elapsed.count() << " seconds." << '\n';
24
25     co_return;
26 }
```

```
27
28 void producer() {
29
30     using namespace std::chrono_literals;
31     std::this_thread::sleep_for(2s);
32
33     event.set(); // resumes the consumer
34
35 }
36
37 int main() {
38
39     std::cout << '\n';
40
41     auto con = std::async([]{ cppcoro::sync_wait(consumer()); });
42     auto prod = std::async(producer);
43
44     con.get(), prod.get();
45
46     std::cout << '\n';
47
48 }
```

---

Проанализируем приведенный пример. Функции `consumer` и `producer` выполняются каждая в своем потоке. Вызов `cppcoro::sync_wait(consumer())` (строка 41) работает как высокоуровневая задача, поскольку `main` не может быть сопрограммой. Вызов ожидает, пока сопрограмма `consumer` завершится. Сопрограмма `consumer` ожидает `co_wait event` (строка 19) до тех пор, пока кто-нибудь не вызовет `event.set()` (строка 33). Функция `producer` запускает событие после двух секунд ожидания.

```
1 // cppcoro::single_consumer.cpp
2
3 #include <cppcoro/single_consumer_event.hpp>
4 #include <cppcoro/sync_wait.hpp>
5 #include <cppcoro/task.hpp>
6
7 #include <future>
8 #include <iostream>
9 #include <string>
10 #include <thread>
11 #include <chrono>
12
13 cppcoro::single_consumer_event event;
14
15 cppcoro::task<> consumer() {
```

Однократная синхронизация потоков через `cppcoro`

Библиотека `cppcoro` также поддерживает мьютексы<sup>1</sup>.

---

<sup>1</sup> [https://en.cppreference.com/w/cpp/named\\_req/Mutex](https://en.cppreference.com/w/cpp/named_req/Mutex).

### 8.1.1.2.2 `async_mutex`

Мьютекс, такой как `cppcoro::async_mutex`, – это механизм синхронизации для защиты совместных данных при одновременном доступе к ним нескольких потоков.

Взаимное исключение (mutual exclusion) при помощи `cppcoro`

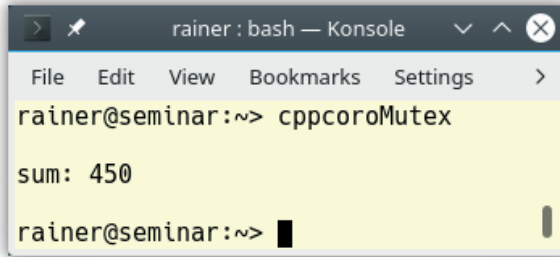
---

```

1  // cppcoroMutex.cpp
2
3  #include <cppcoro/async_mutex.hpp>
4  #include <cppcoro/sync_wait.hpp>
5  #include <cppcoro/task.hpp>
6
7  #include <iostream>
8  #include <thread>
9  #include <vector>
10
11
12  cppcoro::async_mutex mutex;
13
14  int sum{};
15
16  cppcoro::task<> addToSum(int num) {
17      cppcoro::async_mutex_lock lockSum = co_await mutex.scoped_lock_async();
18      sum += num;
19
20  }
21
22  int main() {
23
24      std::cout << '\n';
25
26      std::vector<std::thread> vec(10);
27
28      for(auto& thr: vec) {
29          thr = std::thread([]{
30              for(int n = 0; n < 10; ++n) cppcoro::sync_wait(addToSum(n)); } );
31      }
32
33      for(auto& thr: vec) thr.join();
34
35      std::cout << "sum: " << sum << '\n';
36
37      std::cout << '\n';
38
39  }
```

---

Строка 26 создает 10 потоков. Каждый поток добавляет числа от 0 до 9 к совместно используемой переменной `sum` (строка 14). Функция `addToSum` является сопрограммой. Сопрограмма ожидает в выражении `co_awaitmutex.scored_lock_async()` (строка 17) до получения мьютекса. Сопрограмма, которая ожидает мьютекс, не блокируется, а приостанавливается (*suspended*). Предыдущий инициатор блокировки возобновляет ожидающую сопрограмму в своем вызове разблокировки. Мьютекс остается заблокированным до конца своей области определения (строка 20).



Взаимное исключение при помощи `cppcoro`

### 8.1.1.3 Функции

Для работы с ожидаемыми объектами появились новые интересные функции:

- `sync_wait()`
- `when_all()`
- `when_all_ready()`
- `fmap()`
- `schedule_on()`
- `resume_on()`

Функция `when_all` создает ожидаемый объект, который ждет все свои входные ожидаемые объекты и возвращает последовательность, которая содержит результат каждой задачи в наборе (*aggregate*).

Ожидание всех ожидаемых объектов при помощи `when_all`

```

1  // cppcoroWhenAll.cpp
2
3  #include <chrono>
4  #include <iostream>
5  #include <thread>
6
7  #include <cppcoro/sync_wait.hpp>
8  #include <cppcoro/task.hpp>
9  #include <cppcoro/when_all.hpp>
10
11 using namespace std::chrono_literals;
12
```

---

```

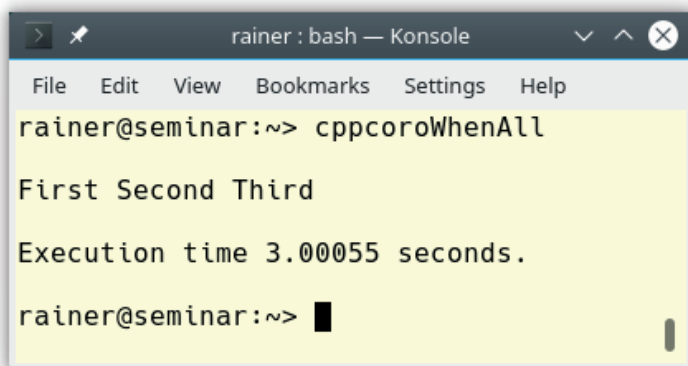
13 cppcoro::task<std::string> getFirst() {
14     std::this_thread::sleep_for(1s);
15     co_return "First";
16 }
17
18 cppcoro::task<std::string> getSecond() {
19     std::this_thread::sleep_for(1s);
20     co_return "Second";
21 }
22
23 cppcoro::task<std::string> getThird() {
24     std::this_thread::sleep_for(1s);
25     co_return "Third";
26 }
27
28
29 cppcoro::task<> runAll() {
30
31     auto [fir, sec, thi] = co_await cppcoro::when_all(getFirst(), getSecond(),
32                                                       getThird());
33
34     std::cout << fir << " " << sec << " " << thi << '\n';
35
36 }
37
38 int main() {
39
40     std::cout << '\n';
41
42     auto start = std::chrono::steady_clock::now();
43
44     cppcoro::sync_wait(runAll());
45
46     std::cout << '\n';
47
48     auto end = std::chrono::high_resolution_clock::now();
49     std::chrono::duration<double> elapsed = end - start;
50     std::cout << "Execution time " << elapsed.count() << " seconds." << '\n';
51
52     std::cout << '\n';
53
54 }

```

---

Задача верхнего уровня `cppcoro::sync_wait(runAll())` (строка 44) ждет ожидаемый объект `runAll`. Он, в свою очередь, ожидает `getFirst`, `getSecond` и `getThird`

(строка 31). Ожидаемые объекты `runAll`, `getFirst`, `getSecond` и `getThird` – это все сопрограммы. Каждая из этих функций спит в течение одной секунды (строки 146, 19 и 24). В сумме получается 3 секунды. Это то время, которое `srpcoro::sync_wait(runAll())` ждет сопрограмма. В строке 49 выводится время выполнения программы.



```

rainer : bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~> cppcoroWhenAll
First Second Third
Execution time 3.00055 seconds.
rainer@seminar:~>

```

Ожидание всех ожидаемых объектов при помощи `when_all`

Вы можете совмещать использование `when_all` с пулами потоков (наборами потоков, `thread pools`) в `srpcoro`.

#### 8.1.1.4 `static_thread_pool`

Класс `static_thread_pool` занимается распределением работы среди пула потоков фиксированного размера.

При создании экземпляра `srpcoro::static_thread_pool` вы можете задать количество потоков в пуле или же не задавать его. Если количество потоков не указано, то для его определения используется функция `std::thread::hardware_concurrency()` из C++11. Функция `std::thread::hardware_concurrency`<sup>1</sup> возвращает подсказку по количеству аппаратно поддерживаемых потоков для аппаратного обеспечения, на котором выполняется программа. Это может быть имеющееся в системе количество процессоров или ядер.

Следующий пример основан на предыдущем примере `srpcoroWhenAll.cpp`, использующем `when_any`. На этот раз все сопрограммы будут выполняться одновременно.

Ожидание одновременно выполняющихся ожидаемых объектов при помощи `when_all`

---

```

1 // cppcoroWhenAllOnThreadPool.cpp
2
3 #include <chrono>
4 #include <iostream>
5 #include <thread>
6

```

<sup>1</sup> [https://en.cppreference.com/w/cpp/thread/thread/hardware\\_concurrency](https://en.cppreference.com/w/cpp/thread/thread/hardware_concurrency).

```

7  #include <cppcoro/sync_wait.hpp>
8  #include <cppcoro/task.hpp>
9  #include <cppcoro/static_thread_pool.hpp>
10 #include <cppcoro/when_all.hpp>
11
12
13 using namespace std::chrono_literals;
14
15 cppcoro::task<std::string> getFirst() {
16     std::this_thread::sleep_for(1s);
17     co_return "First";
18 }
19
20 cppcoro::task<std::string> getSecond() {
21     std::this_thread::sleep_for(1s);
22     co_return "Second";
23 }
24
25 cppcoro::task<std::string> getThird() {
26     std::this_thread::sleep_for(1s);
27     co_return "Third";
28 }
29
30 template <typename Func>
31 cppcoro::task<std::string> runOnThreadPool(cppcoro::static_thread_pool& tp,
32                                           Func func) {
33     co_await tp.schedule();
34     auto res = co_await func();
35     co_return res;
36 }
37
38 cppcoro::task<> runAll(cppcoro::static_thread_pool& tp) {
39
40     auto[fir, sec, thi] = co_await cppcoro::when_all(
41         runOnThreadPool(tp, getFirst()),
42         runOnThreadPool(tp, getSecond()),
43         runOnThreadPool(tp, getThird()));
44
45     std::cout << fir << " " << sec << " " << thi << '\n';
46
47 }
48

```

```
49 int main() {
50
51     std::cout << '\n';
52
53     auto start = std::chrono::steady_clock::now();
54
55     cppcoro::static_thread_pool tp;
56     cppcoro::sync_wait(runAll(tp));
57
58     std::cout << '\n';
59
60     auto end = std::chrono::high_resolution_clock::now();
61     std::chrono::duration<double> elapsed = end - start;
62     std::cout << "Execution time " << elapsed.count() << " seconds." << '\n';
63
64     std::cout << '\n';
65
66 }
```

---

Принципиальным отличием от предыдущего примера является использование пула потоков. В строке 55 создается пул потоков `tp` и используется как аргумент функции `runAll(tp)` (строка 56). Функция `runAll` использует пул потоков для параллельного запуска сопрограмм. В строке 40 агрегируются значения от каждой сопрограммы и присваиваются соответствующим переменным. В итоге функция `main` выполняется не три секунды, как раньше, а всего одну.

```
1 // cppcoroWhenAllOnThreadPool.cpp
2
3 #include <chrono>
4 #include <iostream>
5 #include <thread>
6
7 #include <cppcoro/sync_wait.hpp>
8 #include <cppcoro/task.hpp>
9 #include <cppcoro/static_thread_pool.hpp>
10 #include <cppcoro/when_all.hpp>
11
12
13 using namespace std::chrono_literals;
14
15 cppcoro::task<std::string> getFirst() {
16     std::this_thread::sleep_for(1s);
```

Ожидание трех ожидаемых объектов при помощи `when_all`

## 8.1.2 Модуляризированная стандартная библиотека

Все идет к тому, что скоро не потребуется использовать заголовочные файлы стандартной библиотеки. Microsoft поддерживает модули для всех заголовочных файлов STL в соответствии с предложением P0541<sup>1</sup>. Реализация от Microsoft дает первое впечатление, как может выглядеть модуляризированная

---

<sup>1</sup> <http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2017/p0581r0.pdf>.



стандартная библиотека. Вот что я нашел в публикации Using C++ Modules in Visual Studio 2017<sup>1</sup> из блога команды C++ от Microsoft.

### 8.1.2.1 Модули C++ в Visual Studio 2017

- `std::regex` возвращает содержимое файла `<regex>`.
- `std::filesystem` возвращает содержимое заголовочного файла `<experimental/filesystem>`.
- `std::memory` возвращает содержимое файла `<memory>`.
- `std::threading` возвращает содержимое заголовочных файлов `<atomic>`, `<condition_variable>`, `<future>`, `<mutex>`, `<shared_mutex>` и `<thread>`.
- `std::core` возвращает все остальное из стандартной библиотеки C++.

Для использования модулей для стандартной библиотеки от Microsoft вам нужно задать модель обработки исключений (`/EHsc`) и библиотеку многопоточности (`/MD`). Кроме того, вам нужно использовать флаги `/std:c++latest` и `/experimental:module`.

В разделе по модулям я использовал следующее определение модуля:

Определение модуля с фрагментом глобального модуля

---

```

1  // math1.ixx
2
3  module;
4
5  #include <numeric>
6  #include <vector>
7
8  export module math;
9
10 export int add(int fir, int sec){
11     return fir + sec;
12 }
13
14 export int getProduct(const std::vector<int>& vec) {
15     return std::accumulate(vec.begin(), vec.end(), 1,
                             std::multiplies<int>());

```

Это определение модуля может быть переделано с использованием модуляризированной стандартной библиотеки. Нужно только заменить использование заголовочных файлов `<numeric>` и `<vector>` на модуль `std::core`.

<sup>1</sup> <https://devblogs.microsoft.com/cppblog/cpp-modules-in-visual-studio-2017/>.

Импорт модуля std.core в интерфейсный файл

---

```
// math2.ixx

module;

export module math;

import std.core;

export int add(int fir, int sec){
    return fir + sec;
}

export int getProduct(const std::vector<int>& vec) {
    return std::accumulate(vec.begin(), vec.end(), 1, std::multiplies<int>());
}
```

---

Кроме того, вы должны использовать модуль std.core вместо стандартных заголовочных файлов:

Импорт модуля std.core в пользовательскую программу

---

```
// client2.cpp

import math;
import std.core;

int main() {

    std::cout << '\n';

    std::cout << "add(2000, 20): " << add(2000, 20) << '\n';

    std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    std::cout << "getProduct(myVec): " << getProduct(myVec) << '\n';

    std::cout << '\n';

}
```

---

Результат работы программы такой, как ожидалось:

```

C:\Users\seminar>math2.exe

add(2000, 20): 2020
getProduct(myVec): 3628800

C:\Users\seminar>

```

Использование модуля `std.core` в Windows

### 8.1.3 Исполнители

У исполнителей (executors) довольно длинная история в C++. Их обсуждение началось в 2010 году. Детлеф Воллманн дает их великолепный обзор в своей презентации *Finally Executors for C++*<sup>1</sup>.

Мое введение в исполнители основано на предложении по реализации исполнителей P0761<sup>2</sup> и их формальном описании P0443<sup>3</sup>. Я также ссылаюсь на относительно недавнее предложение P1055<sup>4</sup>.

Что такое исполнители?

Исполнители – это базовые строительные блоки для выполнения в C++. Они играют ту же роль в выполнении, что и аллокатеры в контейнерах C++. Многие предложения по исполнителям уже опубликованы, и многие документы по их реализации все еще открыты. Они должны быть частью C++23, но, скорее всего, могут начать использоваться значительно раньше для расширения стандарта C++.

Исполнитель состоит из правил, где, когда и как выполнять вызываемый объект (callable).

- **Где:** исполнитель может выполняться на внутреннем или внешнем процессоре. Результат читается обратно с внутреннего или внешнего процессора.
- **Когда:** вызываемый объект может выполняться сразу же или же быть поставлен на исполнение.
- **Как:** исполняемый объект может выполняться на CPU или GPU или даже выполняться векторизованно.

Возможности по параллелизму и одновременности (concurrency) в C++ сильно зависят от исполнителей как строительных блоков для выполнения. Эта зависимость справедлива и для таких существующих возможностей, как параллельные алгоритмы из стандартной библиотеки<sup>5</sup>, и для новых возможностей,

<sup>1</sup> <http://www.vollmann.ch/en/presentations/executors2018.pdf>.

<sup>2</sup> <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0761r2.pdf>.

<sup>3</sup> <http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p0443r7.html>.

<sup>4</sup> <http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p1055r0.pdf>.

<sup>5</sup> <https://www.modernescpp.com/index.php/parallel-algorithm-of-the-standard-template-library>.

таких как защелки и барьеры, сопрограммы, сетевая библиотека, расширенные объекты-future<sup>1</sup>, транзакционная память<sup>2</sup> или блоки задач (task blocks)<sup>3</sup>.

### 8.1.3.1 Первые примеры

Следующие примеры должны дать вам первое представление об исполнителях.

#### 8.1.3.1.1 Использование исполнителя

- Объект-обещание `std::async`

`std::async` использует исполнитель

---

```
// get an executor through some means
my_executor_type my_executor = ...

// launch an async using my executor
auto future = std::async(my_executor, [] {
    std::cout << "Hello world, from a new execution agent!" < '\n';
});
```

---

- Алгоритм `std::for_each`

```
// get an executor through some means
my_executor_type my_executor = ...

// execute a parallel for_each "on" my executor
std::for_each(std::execution::par.on(my_executor),
              data.begin(), data.end(), func);
```

---

#### 8.1.3.1.2 Получение исполнителя

Для получения исполнителя существует несколько способов:

- из контекста выполнения `static_thread_pool`;

Исполнитель из `static_thread_pool`

---

```
// create a thread pool with 4 threads
static_thread_pool pool(4);

// get an executor from the thread pool
auto exec = pool.executor();

// use the executor on some long-running task
auto task1 = long_running_task(exec);
```

---

---

<sup>1</sup> <https://www.modernescpp.com/index.php/std-future-extensions>.

<sup>2</sup> <https://www.modernescpp.com/index.php/transactional-memory>.

<sup>3</sup> <https://www.modernescpp.com/index.php/task-blocks>.

- от системного исполнителя.

Системный исполнитель – это используемый по умолчанию исполнитель, при условии что он не был изменен;

- от адаптера исполнителя.

Использование адаптера исполнителя

---

```
// get an executor from a thread pool
auto exec = pool.executor();

// wrap the thread pool's executor in a logging_executor
logging_executor<decltype(exec)> logging_exec(exec);

// use the logging executor in a parallel sort
std::sort(std::execution::par.on(logging_exec), my_data.begin(), my_data.end());
```

---

В этом примере `logging_executor` – это обертка для пула исполнителей.

### 8.1.3.2 Цели концепции исполнителей

Что является целями появления концепции исполнителей в соответствии с предложением P1055<sup>1</sup>? Исполнитель:

- **Группируемый:** управляет балансом между ценой передачи управления исполнителю и его размером.
- **Гетерогенный:** позволяет выполняться в гетерогенном контексте (вычислители однородны) и передавать результат обратно.
- **Упорядочиваемый:** задает порядок, в котором вызываемые (callable) объекты будут выполняться. Эта цель включает в себя такие гарантии, как LIFO (Last In, First Out), FIFO (First In, First Out), приоритеты или ограничения по времени, а также последовательное выполнение.
- **Управляемый:** вызываемый объект можно назначать на конкретное устройство, откладывать выполнение на потом или даже отменять.
- **Продолжаемый:** для неблокирующей передачи исполняемых объектов (work units) требуются сигналы от исполняемых объектов. Эти сигналы должны обозначать, доступен ли результат, случилась ли ошибка, вызываемый объект завершил выполнение или вызывающий решил прервать работу вызываемого. Принудительное начало работы вызываемого или же его остановка также должны быть возможны.
- **Иерархичный:** иерархия позволяет добавлять новые возможности без усложнения простейших случаев.
- **Используемый:** легкость использования как для реализующего, так и для использующего должна быть главной целью.
- **Расширяемый:** пользователи должны иметь возможность расширять исполнителей для добавления возможностей, не являющихся частью стандарта.
- **Минимальный:** в концепции исполнителей не должно быть ничего, что могло бы быть добавлено через библиотеку поверх основного концепта.

<sup>1</sup> <http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p1055r0.pdf>.

### 8.1.3.3 Исполняющая функция

Исполнитель предоставляет одну или несколько функций для создания агентов исполнения из вызываемого объекта. Исполнитель должен поддерживать как минимум один из шести следующих методов.

Методы исполнителя

| Метод                            | Счетность | Направление |
|----------------------------------|-----------|-------------|
| <code>execute</code>             | Single    | Oneway      |
| <code>twoway_execute</code>      | Single    | Twoway      |
| <code>then_execute</code>        | Single    | Then        |
| <code>bulk_execute</code>        | Bulk      | Oneway      |
| <code>bulk_twoway_execute</code> | Bulk      | twoway      |
| <code>bulk_then_execute</code>   | bulk      | Then        |

У каждого метода есть два свойства: счетность и направление.

○ Счетность:

- ♦ `single` – создает одного агента исполнения;
- ♦ `bulk` – создает группу агентов исполнения.

○ Направление:

- ♦ `oneway` – создает агента и не возвращает результат;
- ♦ `twoway` – создает агента и возвращает объект `future`, который может быть использован для ожидания завершения выполнения;
- ♦ `then` – создает агента и возвращает объект `future`, который может быть использован для ожидания завершения выполнения. Агент выполнения начинает выполнение после того, как заданный объект `future` будет готов.

Дадим более строгое определение этих методов.

Для начала рассмотрим единичную счетность:

- метод выполнения `oneway` – это задания вида `fire-and-forget` (запустил и забыл). Очень похоже на объект `future fire-and-forget`, но нет автоматической блокировки в деструкторе объекта-`future`<sup>1</sup>;
- метод `twoway` возвращает вам объект `future`, который можно использовать для получения результата. Ведет себя аналогично `std::promise`<sup>2</sup>, который возвращает ссылку на связанный объект `future`;
- метод `then` возвращает объект `future`. При этом агент исполнения выполняется, только если данный объект `future` готов.

Второй случай счетности (`bulk`) более сложен. Эти методы создают группу агентов исполнения, и каждый из этих агентов вызывает вызываемый объект. Они возвращают результат в виде фабрики, и пользователь сам отвечает за получение правильного результата через эту фабрику.

<sup>1</sup> <https://www.modernescpp.com/index.php/the-special-futures>.

<sup>2</sup> <https://www.modernescpp.com/index.php/promise-and-future>.

### 8.1.3.3.1 `execution::require`

Как можно убедиться, что исполнитель поддерживает заданный метод выполнения?

В отдельных случаях это изначально известно.

Исполнитель, использующий метод `execute`

---

```
void concrete_context(const my_oneway_single_executor& ex)
{
    auto task = ...;
    ex.execute(task);
}
```

---

В более общем случае можно использовать `execution::require`, для того чтобы узнать об этом.

Исполнитель, который должен иметь `single` и `twoway`-функцию

---

```
template <typename Executor>
void generic_context(const Executor& ex)
{
    auto task = ...;

    // ensure .twoway_execute() is available with execution::require()
    execution::require(ex, execution::single,
                      execution::twoway).twoway_execute(task\
);
```

---

В этом случае исполнитель `ex` должен поддерживать счетность `single` и направление `twoway`.

## 8.1.4 Сетевая библиотека

Сетевая библиотека в C++23 основана на библиотеке `boost::asio`<sup>1</sup> от Кристофера Колхоффа. Данная библиотека предназначена для программирования сетевых приложений и низкоуровневого ввода/вывода.

Частями библиотеки являются следующие компоненты:

- TCP, UDP и многоадресная рассылка;
- приложения клиент/сервер;
- масштабируемость на несколько одновременных соединений;
- IPv4 и IPv6;
- разрешение имен (DNS);
- часы.

---

<sup>1</sup> [https://www.boost.org/doc/libs/1\\_75\\_0/doc/html/boost\\_asio.html](https://www.boost.org/doc/libs/1_75_0/doc/html/boost_asio.html).

Следующие компоненты не являются частью сетевой библиотеки:

- реализация сетевых протоколов, таких как HTTP, SMTP или FTP;
- шифрование (TLS или SSL);
- работа с мультиплексирующими интерфейсами, такими как select или poll;
- поддержка работы в реальном времени;
- протоколы TCP/IP, например ICMP.

Благодаря сетевой библиотеке можно легко реализовать простейший эхо-сервер.

Простой эхо-сервер

---

```
1  template <typename Iterator>
2  void uppercase(Iterator begin, Iterator end) {
3      std::locale loc("");
4      for (Iterator iter = begin; iter != end; ++iter)
5          *iter = std::toupper(*iter, loc);
6  }
7
8  void sync_connection(tcp::socket& socket) {
9      try {
10         std::vector<char> buffer_space(1024);
11         while (true) {
12             std::size_t length = socket.read_some(buffer(buffer_space));
13             uppercase(buffer_space.begin(), buffer_space.begin() + length);
14             write(socket, buffer(buffer_space, length));
15         }
16     }
17     catch (std::system_error& e) {
18         // ...
19     }
20 }
```

---

Сервер получает клиентский socket (строка 8), который читает текст (строка 12), преобразует текст в заглавные буквы (строка 13) и посылает текст обратно клиенту (строка 14).

В библиотеке boost есть более сложные примеры чата или HTTP-серверов. Кроме того, сервер может выполняться как синхронно (как в этом примере), так и асинхронно.



## 8.2 C++23 или позже

Еще неизвестно, станут ли следующие три новых нововведения – контракты, рефлексия и сопоставление с образцом – частью C++23. Общая идея в том, что они будут частью будущего стандарта C++. Это значит, что они будут частично поддерживаться в C++23.

### 8.2.1 Контракты

Контракты планировались как пятое большое нововведение в C++20. Но из-за проблем с их разработкой они были убраны комитетом по стандартизации 19 июля 2019 г. в Колонье. В это же самое время была создана группа по контрактам (study group 21 for contracts)<sup>1</sup>.

#### ○ Что такое контракт?

Контракт задается точным и проверенным интерфейсом к программным компонентам. Под программными компонентами обычно подразумеваются функции и методы, которые должны выполнять определенные предварительные условия (preconditions), постусловия (postconditions) или инварианты. Ниже приводятся упрощенные определения этих трех понятий.

- **Предварительные условия:** предикат, который должен выполняться при каждом входе в функцию.
- **Постусловия:** предикат, который должен выполняться при каждом выходе из функции.
- **Инвариант:** предикат, который должен выполняться в своем месте в вычислениях.

Предварительные условия и постусловия помещаются вне определения функции, но инвариант (invariant, assertion) помещается внутри. Предикат – это функция, которая возвращает логическое значение.

Рассмотрим пример использования контрактов.

Функция `push` использует контракты

---

```
int push(queue& q, int val)
    [[ expects: !q.full() ]]
    [[ ensures !q.empty() ]] {
    ...
    [[ assert: q.is_ok() ]]
    ...
}
```

---

Атрибут `expects` – это предварительное условие, атрибут `ensures` – это постусловие, и атрибут `assert` – это инвариант. Контрактами для функции `push` является то, что очередь не заполнена перед добавлением нового элемента, что она не пуста после добавления и справедлив инвариант `q.is_ok()`.

<sup>1</sup> <https://isocpp.org/std/the-committee>.

Предварительные условия и постусловия – это часть интерфейса функции. Это значит, что они не могут обращаться к локальным переменным и к `private`- и `protected`-членам класса. Инварианты при этом являются частью реализации и поэтому могут обращаться к `private`- и `protected`-членам класса.

Обращение к `private`-атрибуту

---

```
class X {  
public:  
    void f(int n)  
        [[ expects: n < m ]] // error; m is private  
    {  
        [[ assert: n < m ]]; // OK  
        // ...  
    }  
private:  
    int m;  
};
```

---

Атрибут `m` является `private` и не может быть частью предварительного условия. По умолчанию нарушение контракта прерывает выполнение программы.

Поведение атрибутов можно настраивать.

### 8.2.1.1 Настройка атрибутов

Синтаксис для настройки атрибутов довольно сложен: `[[contract-attribute modifier: conditional-expression]]`.

- **contract-attribute**: `expects`, `ensures` и `assert`.
- **modifier**: задает уровень проверки выполнения контракта; возможными значениями являются `default`, `audit` и `axiom`.
  - ◆ **default**: цена проверки во время выполнения должна быть небольшая; это параметр по умолчанию.
  - ◆ **audit**: цена проверки во время выполнения предполагается большой.
  - ◆ **axiom**: предикат не проверяется во время выполнения.
- **conditional-expression**: предикат контракта.

Для атрибута `ensures` можно дополнительно указать идентификатор: `[[ensures modifier identifier: conditional-expression]]`.

Идентификатор позволяет сослаться на возвращаемое значение функции.

Доступ к возвращаемому значению

---

```
int mul(int x, int y)  
    [[expects: x > 0]] // implicit default  
    [[expects default: y > 0]]  
    [[ensures audit res: res > 0]] {  
    return x * y;  
}
```

---

Выступающее в качестве идентификатора имя `ges` является произвольным. Как показано в этом примере, в программе может быть несколько контрактов одного и того же типа.

Рассмотрим более подробно обработку нарушений контрактов.

### 8.2.1.2 Обработка нарушений контрактов

У компиляции есть три уровня проверки контрактов:

- `off`: контракты вообще не проверяются;
- `default`: проверяются только `default`-контракты, это выбор по умолчанию;
- `audit`: проверяются контракты `default` и `audit`.

Когда происходит нарушение контракта из-за того, что предикат возвращает `false`, вызывается обработчик нарушений. Обработчик нарушений получает значение типа `std::contract_violation`. Это значение содержит подробную информацию о нарушении контракта.

Класс `contract_violation`

---

```
namespace std {
    class contract_violation{
    public:
        uint_least32_t line_number() const noexcept;
        string_view file_name() const noexcept;
        string_view function_name() const noexcept;
        string_view comment() const noexcept;
        string_view assertion_level() const noexcept;
    };
}
```

---

- `line_number` – номер строки, где произошло нарушение.
- `file_name` – имя файла, где произошло нарушение.
- `function_name` – имя функции, где произошло нарушение.
- `comment` – предикат контракта.
- `assertion_level` – уровень сообщения об ошибке контракта.

### 8.2.1.3 Объявление контрактов

Контракт можно поместить в объявление функции. Это относится к объявлению виртуальных функций и шаблонных функций.

- Объявления контракта функции должны совпадать. Любое объявление, отличное от первого, может привести к пропуску контракта.

Объявления контракта должны совпадать

```
int f(int x)
    [[expects: x > 0]]
    [[ensures r: r > 0]];

int f(int x); // OK. No contract.

int f(int x)
    [[expects: x >= 0]]; // Error missing ensures and different
```

- Контракт не может быть изменен в перегруженной функции.

Перегруженные функции не могут изменить контракт

```
struct B {
    virtual void f(int x) [[expects: x > 0]];
    virtual void g(int x);
};

struct D: B{
    void f(int x) [[expects: x >= 0]]; // error
    void g(int x) [[expects: x != 0]]; // error
};
```

Оба объявления контрактов в классе D ошибочны. Контракт D::f отличается от контракта A::f. Метод D::g добавляет контракт к B::g.



### Заключительные размышления от Герба Саттера

Контракты должны были быть частью C++20, но были отложены как минимум до C++23. Мысли Герба Саттера в Sutter's Mill<sup>1</sup> могут дать некоторое представление об их важности: «контракты – это очень впечатляющая возможность C++20 на настоящее время, и, пожалуй, самая важная функция, которая была добавлена в C++ начиная с C++11».

## 8.2.2 Рефлексия

Рефлексия (reflection) – это возможность программы анализировать и изменять саму себя. Рефлексия происходит во время компиляции и поэтому поддерживается метаправила C++: «не платите за то, что вы не используете». Библиотека признаков типов (type traits library)<sup>2</sup> является мощным инструментом

<sup>1</sup> <https://herbsutter.com/2018/07/02/trip-report-summer-iso-c-standards-meeting-rapperswil/>.

<sup>2</sup> [https://en.cppreference.com/w/cpp/header/type\\_traits](https://en.cppreference.com/w/cpp/header/type_traits).

для рефлексии, но предложение P0385<sup>1</sup> для статической рефлексии идет гораздо дальше.

Следующий пример кода должен дать вам первое представление о рефлексии.

Оператор рефлексии

---

```

1  template <typename T>
2  T min(constT& a, constT& b) {
3      log() << "function: min<"
4          << get_base_name_v<get_aliased_t<$reflect(T)>>
5          << ">("
6          << get_base_name_v<$reflect(a)> << ": "
7          << get_base_name_v<get_aliased_t<get_type_t<$reflect(a)>>>
8          << " = " << a << ", "
9          << get_base_name_v<$reflect(b)> << ": "
10         << get_base_name_v<get_aliased_t<get_type_t<$reflect(b)>>>
11         << " = " << b
12         << ")" << '\n';
13     return a < b ? a : b;
14 }
```

---

Новый оператор рефлексии `$reflect` является самым важным выражением в этом примере. Этот новый оператор создает специальный тип данных, который предоставляет метаинформацию о шаблонном параметре `T` (строка 4) и значениях `a` (строка 6) и `b` (строка 9). Также метаинформация может быть использована для получения дополнительной информации: `et_base_name_v<get_aliased_t ....` (строки 7 и 10).

При вызове функции `min` с аргументами `min(12.34, 23.45)` будет следующий вывод:

```

1  template <typename T>
2  T min(constT& a, constT& b) {
```

Вызов `min(12.34, 23.45)`

Разберемся, что за метаинформацию можно получить при помощи рефлексии. Это:

- объекты (objects): строка в исходном файле и имя файла;
- классы (classes): `private`- и `public`-члены класса и методы;
- псевдонимы (aliases): имена разрешенных псевдонимов.

Следующий пример из предложения P0385 показывает, как рефлексия помогает определить `private`- и `public`-члены класса.

---

<sup>1</sup> <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0385r2.pdf>.

Определение private- и public-членов класса foo

---

```
#include <reflect>
#include <iostream>

struct foo {
    private:
        int _i, _j;
    public:
        static constexpr const bool b = true;
        float x, y, z;
    private:
        static double d;
};

template <typename ... T>
void eat(T ... ) { }

template <typename Metaobjects, std::size_t I>
int do_print_data_member(void) {
    using namespace std;
    typedef reflect::get_element_t<Metaobjects, I> metaobj;
    cout << I << ": "
        << (reflect::is_public_v<metaobj>?"public":"non-public")
        << " "
        << (reflect::is_static_v<metaobj>?"static":"" )
        << " "
        << reflect::get_base_name_v<reflect::get_type_t<metaobj>>
        << " "
        << reflect::get_base_name_v<metaobj>
        << '\n';
}

return 0;

template <typename Metaobjects, std::size_t ... I>
void do_print_data_members(std::index_sequence<I...>) {
    eat(do_print_data_member<Metaobjects, I>()...);
}

template <typename Metaobjects>
void do_print_data_members(void) {
    using namespace std;
```

---

```

        do_print_data_members<Metaobjects>(
            make_index_sequence<
                reflect::get_size_v<Metaobjects>
            >()
        );
    }

    template <typename MetaClass>
    void print_data_members(void) {
        using namespace std;

        cout << "Public data members of " << reflect::get_base_name_v<MetaClass>
            << '\n';

        do_print_data_members<reflect::get_public_data_members_t<MetaClass>>();
    }

    template <typename MetaClass>
    void print_all_data_members(void) {
        using namespace std;

        cout << "All data members of " << reflect::get_base_name_v<MetaClass>
            << '\n';
        do_print_data_members<reflect::get_data_members_t<MetaClass>>();
    }

    int main(void) {
        print_data_members<$reflect(foo)>();
        print_all_data_members<$reflect(foo)>();
        return 0;
    }

```

---

Данная программа производит следующий вывод:

```
Public data members of foo
0: public static bool b
1: public float x
2: public float y
3: public float z
All data members of foo
0: non-public int _i
1: non-public int _j
2: public static bool b
3: public float x
4: public float y
5: public float z
6: non-public static double d
```

Показ public- и private-членов класса foo

### 8.2.3 Сопоставление с образцом

Новым типам данных, таким как `std::tuple`<sup>1</sup> и `std::variant`<sup>2</sup>, нужны новые способы работы с их элементами. Простые условия вроде `if` или `switch` или функции вроде `std::apply`<sup>3</sup> или `std::visit`<sup>4</sup> предоставляют только базовую функциональность. Сопоставление с образцом (соответствие шаблону, Pattern Matching), широко используемое в функциональном программировании, дает более удобные способы обработки новых типов данных.

Следующий фрагмент кода из предложения P1371R2<sup>5</sup> по соответствию шаблонов сравнивает классические управляющие конструкции и сопоставление с образцом. Операция сопоставления с образцом использует ключевое слово `inspect` и `__` для обозначения заполнителя (placeholder).

- Оператор `switch`.

---

<sup>1</sup> <https://en.cppreference.com/w/cpp/utility/tuple>.

<sup>2</sup> <https://en.cppreference.com/w/cpp/utility/variant>.

<sup>3</sup> <https://en.cppreference.com/w/cpp/utility/apply>.

<sup>4</sup> <https://en.cppreference.com/w/cpp/utility/variant/visit>.

<sup>5</sup> <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1371r2.pdf>.



---

Оператор `switch` по сравнению с сопоставлением с образцом

---

```
switch (x) {
    case 0: std::cout << "got zero"; break;
    case 1: std::cout << "got one"; break;
    default: std::cout << "don't care";
}
```

```
inspect (x) {
    0: std::cout << "got zero";
    1: std::cout << "got one";
    __: std::cout << "don't care";
}
```

---

#### ○ Условие `if`.

Оператор `if` по сравнению с сопоставлением с образцом

---

```
if (s == "foo") {
    std::cout << "got foo";
} else if (s == "bar") {
    std::cout << "got bar";
} else {
    std::cout << "don't care";
}
```

```
inspect (s) {
    "foo": std::cout << "got foo";
    "bar": std::cout << "got bar";
    __: std::cout << "don't care";
}
```

---

Применение сопоставления с образцом к `std::tuple` и `std::variant` или полиморфным объектам демонстрирует всю силу этого подхода.

#### ○ `std::tuple`.

`std::tuple` и сопоставление с образцом

---

```
auto&& [x, y] = p;
if (x == 0 && y == 0) {
    std::cout << "on origin";
} else if (x == 0) {
    std::cout << "on y-axis";
} else if (y == 0) {
    std::cout << "on x-axis";
} else {
    std::cout << x << ',' << y;
}

inspect (p) {
    [0, 0]: std::cout << "on origin";
    [0, y]: std::cout << "on y-axis";
    [x, 0]: std::cout << "on x-axis";
    [x, y]: std::cout << x << ',' << y;
}
```

---

○ `std::variant`.

`std::variant` и сопоставление с образцом

---

```
struct visitor {
    void operator()(int i) const {
        os << "got int: " << i;
    }
    void operator()(float f) const {
        os << "got float: " << f;
    }
    std::ostream& os;
};

std::visit(visitor{strm}, v);

inspect (v) {
    <int> i: strm << "got int: " << i;
    <float> f: strm << "got float: " << f;
}
```

---

○ Полиморфные типы данных.

Полиморфные типы данных и сопоставление с образцом

---

```

struct Shape { virtual ~Shape() = default; };
struct Circle : Shape { int radius; };
struct Rectangle : Shape { int width, height; };

virtual int Shape::get_area() const = 0;

int Circle::get_area() const override {
    return 3.14 * radius * radius;
}
int Rectangle::get_area() const override {
    return width * height;
}

int get_area(const Shape& shape) {
    return inspect (shape) {
        <Circle> [r] => 3.14 * r * r,
        <Rectangle> [w, h] => w * h
    }
}

```

---

Предложение P1371R2 по введению операции сопоставления с образцом предлагает и более продвинутые примеры его использования. Например, сопоставление с образцом может использоваться для обхода дерева выражения<sup>1</sup>.

---

<sup>1</sup> [https://en.wikipedia.org/wiki/Binary\\_expression\\_tree](https://en.wikipedia.org/wiki/Binary_expression_tree).

## 8.3 Дополнительная информация о стандарте C++23

Предложение P0592R4<sup>1</sup> описывает общую идею нововведений в стандарте C++23 и описывает основные идеи. Такие возможности, как блоки задач (task blocks)<sup>2</sup>, унифицированные объекты future<sup>3</sup>, транзакционная память<sup>4</sup> или параллельная библиотека для работы с векторами (data-parallel vector library)<sup>5</sup>, поддерживающая SIMD<sup>6</sup>, даже не были упомянуты. Если вы хотите получить больше информации о будущем C++, то вам будет очень полезно ознакомиться с [cppreference.com/compiler\\_support](https://en.cppreference.com/compiler_support)<sup>7</sup> и документами от комитета по стандартизации, относящимися к C++23<sup>8</sup>.

---

<sup>1</sup> <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0592r4.html>.

<sup>2</sup> <https://www.modernescpp.com/index.php/task-blocks>.

<sup>3</sup> <https://www.modernescpp.com/index.php/the-end-of-the-detour-unified-futures>.

<sup>4</sup> <https://www.modernescpp.com/index.php/transactional-memory>.

<sup>5</sup> <https://en.cppreference.com/w/cpp/experimental/simd>.

<sup>6</sup> <https://en.wikipedia.org/wiki/SIMD>.

<sup>7</sup> [https://en.cppreference.com/w/cpp/compiler\\_support](https://en.cppreference.com/w/cpp/compiler_support).

<sup>8</sup> <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/>.

## 9. Дополнительное тестирование

Заголовочный файл `<version>` позволяет узнать у вашего компилятора его поддержке C++11 и более поздних версий. Вы можете узнать про атрибуты, возможности самого языка или библиотеки. В `<version>` содержится около 200 макросов, получающих числовое значение, когда та или иная возможность поддерживается. Это число задает год и месяц, когда эта возможность была добавлена в стандарт C++. Ниже приводятся номера для `static_assert`, лямбд и концептов.

Макросы для `static_assert`, лямбд и концептов

---

```
__cpp_static_assert  200410L
__cpp_lambdas        200907L
__cpp_concepts       201907L
```

---



### Поддержка функциональности

Когда я экспериментирую с новой функциональностью C++, то проверяю, какие компиляторы поддерживают ту функциональность, которая мне интересна. Для этого я посещаю [cppreference.com/compiler\\_support](https://en.cppreference.com/w/cpp/compiler_support)<sup>1</sup>, ищу ту функциональность, которая мне нужна, и надеюсь, что как минимум один компилятор из трех основных (GCC, Clang, MSVC) реализует ее.

Получение ответа `partial` неудовлетворительно. Если компиляция новой возможности заканчивается ошибкой, то обращаться не к кому.

---

<sup>1</sup> [https://en.cppreference.com/w/cpp/compiler\\_support](https://en.cppreference.com/w/cpp/compiler_support).

| C++20 feature                                                      | Paper(s)           | GCC                           | Clang                   | MSVC                 | Apple Clang | EDG ecpp | Intel C++ | IBM XL C++ | Sun/Oracle C++ | Embarcadero C++ Builder | Cray | Portland Group (PGI) | Nvidia nvcc | [Collapse] |
|--------------------------------------------------------------------|--------------------|-------------------------------|-------------------------|----------------------|-------------|----------|-----------|------------|----------------|-------------------------|------|----------------------|-------------|------------|
| Allow lambda-capture [=, this]                                     | P0409R2            | 8                             | 6                       | 19.22*               | 10.0.0*     | 5.1      |           |            |                |                         |      |                      |             |            |
| __VA_OPT__                                                         | P0306R4<br>P1042R1 | 8 (partial)*<br>10 (partial)* | 9                       | 19.25*               | 11.0.3*     | 5.1      |           |            |                |                         |      |                      |             |            |
| Designated initializers                                            | P0329R4            | 4.7 (partial)*<br>8           | 3.0<br>(partial)*<br>10 | 19.21*               | (partial)*  | 5.1      |           |            |                |                         |      |                      |             |            |
| template-parameter-list for generic lambdas                        | P0428R2            | 8                             | 9                       | 19.22*               | 11.0.0*     | 5.1      |           |            |                |                         |      |                      |             |            |
| Default member initializers for bit-fields                         | P0683R1            | 8                             | 6                       | 19.25*               | 10.0.0*     | 5.1      |           |            |                |                         |      |                      |             |            |
| Initializer list constructors in class template argument deduction | P0702R1            | 8                             | 6                       | 19.14*               | Yes         | 5.0      |           |            |                |                         |      |                      |             |            |
| constexpr-qualified pointers to members                            | P0704R1            | 8                             | 6                       | 19.0*                | 10.0.0*     | 5.1      |           |            |                |                         |      |                      |             |            |
| Concepts                                                           | P0734R0            | 6<br>(TS only)<br>10          | 10                      | 19.23*<br>(partial)* |             | 6.1      |           |            |                |                         |      |                      |             |            |
| Lambdas in unevaluated contexts                                    | P0315R4            | 9                             |                         | 19.28*               |             |          |           |            |                |                         |      |                      |             |            |

### Поддержка возможностей языка C++20

Страница с [cppreference.com](https://en.cppreference.com/w/cpp/feature_test) по тестированию той или иной функциональности языка<sup>1</sup> использует все макросы в одном очень длинном исходном файле.

Использование всех макросов функциональности языка C++

| C++20 feature                                                      | Paper(s)           | GCC                           | Clang                   | MSVC                 | Apple Clang | EDG ecpp | Intel C++ | IBM XL C++ | Sun/Oracle C++ | Embarcadero C++ Builder | Cray | Portland Group (PGI) | Nvidia nvcc | [Collapse] |
|--------------------------------------------------------------------|--------------------|-------------------------------|-------------------------|----------------------|-------------|----------|-----------|------------|----------------|-------------------------|------|----------------------|-------------|------------|
| Allow lambda-capture [=, this]                                     | P0409R2            | 8                             | 6                       | 19.22*               | 10.0.0*     | 5.1      |           |            |                |                         |      |                      |             |            |
| __VA_OPT__                                                         | P0306R4<br>P1042R1 | 8 (partial)*<br>10 (partial)* | 9                       | 19.25*               | 11.0.3*     | 5.1      |           |            |                |                         |      |                      |             |            |
| Designated initializers                                            | P0329R4            | 4.7 (partial)*<br>8           | 3.0<br>(partial)*<br>10 | 19.21*               | (partial)*  | 5.1      |           |            |                |                         |      |                      |             |            |
| template-parameter-list for generic lambdas                        | P0428R2            | 8                             | 9                       | 19.22*               | 11.0.0*     | 5.1      |           |            |                |                         |      |                      |             |            |
| Default member initializers for bit-fields                         | P0683R1            | 8                             | 6                       | 19.25*               | 10.0.0*     | 5.1      |           |            |                |                         |      |                      |             |            |
| Initializer list constructors in class template argument deduction | P0702R1            | 8                             | 6                       | 19.14*               | Yes         | 5.0      |           |            |                |                         |      |                      |             |            |
| constexpr-qualified pointers to members                            | P0704R1            | 8                             | 6                       | 19.0*                | 10.0.0*     | 5.1      |           |            |                |                         |      |                      |             |            |
| Concepts                                                           | P0734R0            | 6<br>(TS only)<br>10          | 10                      | 19.23*<br>(partial)* |             | 6.1      |           |            |                |                         |      |                      |             |            |
| Lambdas in unevaluated contexts                                    | P0315R4            | 9                             |                         | 19.28*               |             |          |           |            |                |                         |      |                      |             |            |

<sup>1</sup> [https://en.cppreference.com/w/cpp/feature\\_test](https://en.cppreference.com/w/cpp/feature_test).

```

23 #ifdef __has_cpp_attribute
24 # define COMPILER_ATTRIBUTE_VALUE_AS_STRING(s) #s
25 # define COMPILER_ATTRIBUTE_AS_NUMBER(x) COMPILER_ATTRIBUTE_VALUE_AS_STRING(x)
26 # define COMPILER_ATTRIBUTE_ENTRY(attr) \
27     { #attr, COMPILER_ATTRIBUTE_AS_NUMBER(__has_cpp_attribute(attr)) },
28 #else
29 # define COMPILER_ATTRIBUTE_ENTRY(attr) { #attr, "_" },
30 #endif
31
32 // Change these options to print out only necessary info.
33 static struct PrintOptions {
34     constexpr static bool titles           = 1;
35     constexpr static bool attributes       = 1;
36     constexpr static bool general_features = 1;
37     constexpr static bool core_features   = 1;
38     constexpr static bool lib_features     = 1;
39     constexpr static bool supported_features = 1;
40     constexpr static bool unsupported_features = 1;
41     constexpr static bool sorted_by_value  = 0;
42     constexpr static bool cxx11           = 1;
43     constexpr static bool cxx14           = 1;
44     constexpr static bool cxx17           = 1;
45     constexpr static bool cxx20           = 1;
46     constexpr static bool cxx23           = 0;
47 } print;
48
49 struct CompilerFeature {
50     CompilerFeature(const char* name = nullptr, const char* value = nullptr)
51         : name(name), value(value) {}
52     const char* name; const char* value;
53 };
54
55 static CompilerFeature cxx[] = {
56     COMPILER_FEATURE_ENTRY(__cplusplus)
57     COMPILER_FEATURE_ENTRY(__cpp_exceptions)
58     COMPILER_FEATURE_ENTRY(__cpp_rtti)
59 #if 0
60     COMPILER_FEATURE_ENTRY(__GNUC__)
61     COMPILER_FEATURE_ENTRY(__GNUC_MINOR__)
62     COMPILER_FEATURE_ENTRY(__GNUC_PATCHLEVEL__)
63     COMPILER_FEATURE_ENTRY(__GNUG__)
64     COMPILER_FEATURE_ENTRY(__clang__)
65     COMPILER_FEATURE_ENTRY(__clang_major__)

```

```
66 COMPILER_FEATURE_ENTRY(__clang_minor__)
67 COMPILER_FEATURE_ENTRY(__clang_patchlevel__)
68 #endif
69 };
70 static CompilerFeature cxx11[] = {
71 COMPILER_FEATURE_ENTRY(__cpp_alias_templates)
72 COMPILER_FEATURE_ENTRY(__cpp_attributes)
73 COMPILER_FEATURE_ENTRY(__cpp_constexpr)
74 COMPILER_FEATURE_ENTRY(__cpp_decltype)
75 COMPILER_FEATURE_ENTRY(__cpp_delegating_constructors)
76 COMPILER_FEATURE_ENTRY(__cpp_inheriting_constructors)
77 COMPILER_FEATURE_ENTRY(__cpp_initializer_lists)
78 COMPILER_FEATURE_ENTRY(__cpp_lambdas)
79 COMPILER_FEATURE_ENTRY(__cpp_nsdmi)
80 COMPILER_FEATURE_ENTRY(__cpp_range_based_for)
81 COMPILER_FEATURE_ENTRY(__cpp_raw_strings)
82 COMPILER_FEATURE_ENTRY(__cpp_ref_qualifiers)
83 COMPILER_FEATURE_ENTRY(__cpp_rvalue_references)
84 COMPILER_FEATURE_ENTRY(__cpp_static_assert)
85 COMPILER_FEATURE_ENTRY(__cpp_threadsafe_static_init)
86 COMPILER_FEATURE_ENTRY(__cpp_unicode_characters)
87 COMPILER_FEATURE_ENTRY(__cpp_unicode_literals)
88 COMPILER_FEATURE_ENTRY(__cpp_user_defined_literals)
89 COMPILER_FEATURE_ENTRY(__cpp_variadic_templates)
90 };
91 static CompilerFeature cxx14[] = {
92 COMPILER_FEATURE_ENTRY(__cpp_aggregate_nsdmi)
93 COMPILER_FEATURE_ENTRY(__cpp_binary_literals)
94 COMPILER_FEATURE_ENTRY(__cpp_constexpr)
95 COMPILER_FEATURE_ENTRY(__cpp_decltype_auto)
96 COMPILER_FEATURE_ENTRY(__cpp_generic_lambdas)
97 COMPILER_FEATURE_ENTRY(__cpp_init_captures)
98 COMPILER_FEATURE_ENTRY(__cpp_return_type_deduction)
99 COMPILER_FEATURE_ENTRY(__cpp_sized_deallocation)
100 COMPILER_FEATURE_ENTRY(__cpp_variable_templates)
101 };
102 static CompilerFeature cxx14lib[] = {
103 COMPILER_FEATURE_ENTRY(__cpp_lib_chrono_udls)
104 COMPILER_FEATURE_ENTRY(__cpp_lib_complex_udls)
105 COMPILER_FEATURE_ENTRY(__cpp_lib_exchange_function)
106 COMPILER_FEATURE_ENTRY(__cpp_lib_generic_associative_lookup)
107 COMPILER_FEATURE_ENTRY(__cpp_lib_integer_sequence)
108 COMPILER_FEATURE_ENTRY(__cpp_lib_integral_constant_callable)
```



```

109 COMPILER_FEATURE_ENTRY(__cpp_lib_is_final)
110 COMPILER_FEATURE_ENTRY(__cpp_lib_is_null_pointer)
111 COMPILER_FEATURE_ENTRY(__cpp_lib_make_reverse_iterator)
112 COMPILER_FEATURE_ENTRY(__cpp_lib_make_unique)
113 COMPILER_FEATURE_ENTRY(__cpp_lib_null_iterators)
114 COMPILER_FEATURE_ENTRY(__cpp_lib_quoted_string_io)
115 COMPILER_FEATURE_ENTRY(__cpp_lib_result_of_sfinae)
116 COMPILER_FEATURE_ENTRY(__cpp_lib_robust_nonmodifying_seq_ops)
117 COMPILER_FEATURE_ENTRY(__cpp_lib_shared_timed_mutex)
118 COMPILER_FEATURE_ENTRY(__cpp_lib_string_udls)
119 COMPILER_FEATURE_ENTRY(__cpp_lib_transformation_trait_aliases)
120 COMPILER_FEATURE_ENTRY(__cpp_lib_transparent_operators)
121 COMPILER_FEATURE_ENTRY(__cpp_lib_tuple_element_t)
122 COMPILER_FEATURE_ENTRY(__cpp_lib_tuples_by_type)
123 };
124
125 static CompilerFeature cxx17[] = {
126 COMPILER_FEATURE_ENTRY(__cpp_aggregate_bases)
127 COMPILER_FEATURE_ENTRY(__cpp_aligned_new)
128 COMPILER_FEATURE_ENTRY(__cpp_capture_star_this)
129 COMPILER_FEATURE_ENTRY(__cpp_constexpr)
130 COMPILER_FEATURE_ENTRY(__cpp_deduction_guides)
131 COMPILER_FEATURE_ENTRY(__cpp_enumerator_attributes)
132 COMPILER_FEATURE_ENTRY(__cpp_fold_expressions)
133 COMPILER_FEATURE_ENTRY(__cpp_guaranteed_copy_elision)
134 COMPILER_FEATURE_ENTRY(__cpp_hex_float)
135 COMPILER_FEATURE_ENTRY(__cpp_if_constexpr)
136 COMPILER_FEATURE_ENTRY(__cpp_inheriting_constructors)
137 COMPILER_FEATURE_ENTRY(__cpp_inline_variables)
138 COMPILER_FEATURE_ENTRY(__cpp_namespace_attributes)
139 COMPILER_FEATURE_ENTRY(__cpp_noexcept_function_type)
140 COMPILER_FEATURE_ENTRY(__cpp_nontype_template_args)
141 COMPILER_FEATURE_ENTRY(__cpp_nontype_template_parameter_auto)
142 COMPILER_FEATURE_ENTRY(__cpp_range_based_for)
143 COMPILER_FEATURE_ENTRY(__cpp_static_assert)
144 COMPILER_FEATURE_ENTRY(__cpp_structured_bindings)
145 COMPILER_FEATURE_ENTRY(__cpp_template_template_args)
146 COMPILER_FEATURE_ENTRY(__cpp_variadic_using)
147 };
148 static CompilerFeature cxx17lib[] = {
149 COMPILER_FEATURE_ENTRY(__cpp_lib_addressof_constexpr)
150 COMPILER_FEATURE_ENTRY(__cpp_lib_allocator_traits_is_always_equal)
151 COMPILER_FEATURE_ENTRY(__cpp_lib_any)

```

```
152 COMPILER_FEATURE_ENTRY(__cpp_lib_apply)
153 COMPILER_FEATURE_ENTRY(__cpp_lib_array_constexpr)
154 COMPILER_FEATURE_ENTRY(__cpp_lib_as_const)
155 COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_is_always_lock_free)
156 COMPILER_FEATURE_ENTRY(__cpp_lib_bool_constant)
157 COMPILER_FEATURE_ENTRY(__cpp_lib_boyer_moore_searcher)
158 COMPILER_FEATURE_ENTRY(__cpp_lib_byte)
159 COMPILER_FEATURE_ENTRY(__cpp_lib_chrono)
160 COMPILER_FEATURE_ENTRY(__cpp_lib_clamp)
161 COMPILER_FEATURE_ENTRY(__cpp_lib_enable_shared_from_this)
162 COMPILER_FEATURE_ENTRY(__cpp_lib_execution)
163 COMPILER_FEATURE_ENTRY(__cpp_lib_filesystem)
164 COMPILER_FEATURE_ENTRY(__cpp_lib_gcd_lcm)
165 COMPILER_FEATURE_ENTRY(__cpp_lib_hardware_interference_size)
166 COMPILER_FEATURE_ENTRY(__cpp_lib_has_unique_object_representations)
167 COMPILER_FEATURE_ENTRY(__cpp_lib_hypot)
168 COMPILER_FEATURE_ENTRY(__cpp_lib_incomplete_container_elements)
169 COMPILER_FEATURE_ENTRY(__cpp_lib_invoke)
170 COMPILER_FEATURE_ENTRY(__cpp_lib_is_aggregate)
171 COMPILER_FEATURE_ENTRY(__cpp_lib_is_invocable)
172 COMPILER_FEATURE_ENTRY(__cpp_lib_is_swappable)
173 COMPILER_FEATURE_ENTRY(__cpp_lib_laundry)
174 COMPILER_FEATURE_ENTRY(__cpp_lib_logical_traits)
175 COMPILER_FEATURE_ENTRY(__cpp_lib_make_from_tuple)
176 COMPILER_FEATURE_ENTRY(__cpp_lib_map_try_emplace)
177 COMPILER_FEATURE_ENTRY(__cpp_lib_math_special_functions)
178 COMPILER_FEATURE_ENTRY(__cpp_lib_memory_resource)
179 COMPILER_FEATURE_ENTRY(__cpp_lib_node_extract)
180 COMPILER_FEATURE_ENTRY(__cpp_lib_nonmember_container_access)
181 COMPILER_FEATURE_ENTRY(__cpp_lib_not_fn)
182 COMPILER_FEATURE_ENTRY(__cpp_lib_optional)
183 COMPILER_FEATURE_ENTRY(__cpp_lib_parallel_algorithm)
184 COMPILER_FEATURE_ENTRY(__cpp_lib_raw_memory_algorithms)
185 COMPILER_FEATURE_ENTRY(__cpp_lib_sample)
186 COMPILER_FEATURE_ENTRY(__cpp_lib_scoped_lock)
187 COMPILER_FEATURE_ENTRY(__cpp_lib_shared_mutex)
188 COMPILER_FEATURE_ENTRY(__cpp_lib_shared_ptr_arrays)
189 COMPILER_FEATURE_ENTRY(__cpp_lib_shared_ptr_weak_type)
190 COMPILER_FEATURE_ENTRY(__cpp_lib_string_view)
191 COMPILER_FEATURE_ENTRY(__cpp_lib_to_chars)
192 COMPILER_FEATURE_ENTRY(__cpp_lib_transparent_operators)
193 COMPILER_FEATURE_ENTRY(__cpp_lib_type_trait_variable_templates)
194 COMPILER_FEATURE_ENTRY(__cpp_lib_uncaught_exceptions)
```

```

195 COMPILER_FEATURE_ENTRY(__cpp_lib_unordered_map_try_emplace)
196 COMPILER_FEATURE_ENTRY(__cpp_lib_variant)
197 COMPILER_FEATURE_ENTRY(__cpp_lib_void_t)
198 };
199
200 static CompilerFeature cxx20[] = {
201 COMPILER_FEATURE_ENTRY(__cpp_aggregate_paren_init)
202 COMPILER_FEATURE_ENTRY(__cpp_char8_t)
203 COMPILER_FEATURE_ENTRY(__cpp_concepts)
204 COMPILER_FEATURE_ENTRY(__cpp_conditional_explicit)
205 COMPILER_FEATURE_ENTRY(__cpp_consteval)
206 COMPILER_FEATURE_ENTRY(__cpp_constexpr)
207 COMPILER_FEATURE_ENTRY(__cpp_constexpr_dynamic_alloc)
208 COMPILER_FEATURE_ENTRY(__cpp_constexpr_in_decltype)
209 COMPILER_FEATURE_ENTRY(__cpp_constinit)
210 COMPILER_FEATURE_ENTRY(__cpp_deduction_guides)
211 COMPILER_FEATURE_ENTRY(__cpp_designated_initializers)
212 COMPILER_FEATURE_ENTRY(__cpp_generic_lambdas)
213 COMPILER_FEATURE_ENTRY(__cpp_impl_coroutine)
214 COMPILER_FEATURE_ENTRY(__cpp_impl_destroying_delete)
215 COMPILER_FEATURE_ENTRY(__cpp_impl_three_way_comparison)
216 COMPILER_FEATURE_ENTRY(__cpp_init_captures)
217 COMPILER_FEATURE_ENTRY(__cpp_modules)
218 COMPILER_FEATURE_ENTRY(__cpp_nontype_template_args)
219 COMPILER_FEATURE_ENTRY(__cpp_using_enum)
220 };
221 static CompilerFeature cxx20lib[] = {
222 COMPILER_FEATURE_ENTRY(__cpp_lib_array_constexpr)
223 COMPILER_FEATURE_ENTRY(__cpp_lib_assume_aligned)
224 COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_flag_test)
225 COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_float)
226 COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_lock_free_type_aliases)
227 COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_ref)
228 COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_shared_ptr)
229 COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_value_initialization)
230 COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_wait)
231 COMPILER_FEATURE_ENTRY(__cpp_lib_barrier)
232 COMPILER_FEATURE_ENTRY(__cpp_lib_bind_front)
233 COMPILER_FEATURE_ENTRY(__cpp_lib_bit_cast)
234 COMPILER_FEATURE_ENTRY(__cpp_lib_bitops)
235 COMPILER_FEATURE_ENTRY(__cpp_lib_bounded_array_traits)
236 COMPILER_FEATURE_ENTRY(__cpp_lib_char8_t)
237 COMPILER_FEATURE_ENTRY(__cpp_lib_chrono)

```

```
238 COMPILER_FEATURE_ENTRY(__cpp_lib_concepts)
239 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_algorithms)
240 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_complex)
241 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_dynamic_alloc)
242 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_functional)
243 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_iterator)
244 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_memory)
245 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_numeric)
246 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_string)
247 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_string_view)
248 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_tuple)
249 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_utility)
250 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_vector)
251 COMPILER_FEATURE_ENTRY(__cpp_lib_coroutine)
252 COMPILER_FEATURE_ENTRY(__cpp_lib_destroying_delete)
253 COMPILER_FEATURE_ENTRY(__cpp_lib_endian)
254 COMPILER_FEATURE_ENTRY(__cpp_lib_erase_if)
255 COMPILER_FEATURE_ENTRY(__cpp_lib_execution)
256 COMPILER_FEATURE_ENTRY(__cpp_lib_format)
257 COMPILER_FEATURE_ENTRY(__cpp_lib_generic_unordered_lookup)
258 COMPILER_FEATURE_ENTRY(__cpp_lib_int_pow2)
259 COMPILER_FEATURE_ENTRY(__cpp_lib_integer_comparison_functions)
260 COMPILER_FEATURE_ENTRY(__cpp_lib_interpolate)
261 COMPILER_FEATURE_ENTRY(__cpp_lib_is_constant_evaluated)
262 COMPILER_FEATURE_ENTRY(__cpp_lib_is_layout_compatible)
263 COMPILER_FEATURE_ENTRY(__cpp_lib_is_nothrow_convertible)
264 COMPILER_FEATURE_ENTRY(__cpp_lib_is_pointer_interconvertible)
265 COMPILER_FEATURE_ENTRY(__cpp_lib_jthread)
266 COMPILER_FEATURE_ENTRY(__cpp_lib_latch)
267 COMPILER_FEATURE_ENTRY(__cpp_lib_list_remove_return_type)
268 COMPILER_FEATURE_ENTRY(__cpp_lib_math_constants)
269 COMPILER_FEATURE_ENTRY(__cpp_lib_polymorphic_allocator)
270 COMPILER_FEATURE_ENTRY(__cpp_lib_ranges)
271 COMPILER_FEATURE_ENTRY(__cpp_lib_remove_cvref)
272 COMPILER_FEATURE_ENTRY(__cpp_lib_semaphore)
273 COMPILER_FEATURE_ENTRY(__cpp_lib_shared_ptr_arrays)
274 COMPILER_FEATURE_ENTRY(__cpp_lib_shift)
275 COMPILER_FEATURE_ENTRY(__cpp_lib_smart_ptr_for_overwrite)
276 COMPILER_FEATURE_ENTRY(__cpp_lib_source_location)
277 COMPILER_FEATURE_ENTRY(__cpp_lib_span)
278 COMPILER_FEATURE_ENTRY(__cpp_lib_ssize)
279 COMPILER_FEATURE_ENTRY(__cpp_lib_starts_ends_with)
280 COMPILER_FEATURE_ENTRY(__cpp_lib_string_view)
```

```

281 COMPILER_FEATURE_ENTRY(__cpp_lib_syncbuf)
282 COMPILER_FEATURE_ENTRY(__cpp_lib_three_way_comparison)
283 COMPILER_FEATURE_ENTRY(__cpp_lib_to_address)
284 COMPILER_FEATURE_ENTRY(__cpp_lib_to_array)
285 COMPILER_FEATURE_ENTRY(__cpp_lib_type_identity)
286 COMPILER_FEATURE_ENTRY(__cpp_lib_unwrap_ref)
287 };
288
289 static CompilerFeature cxx23[] = {
290     COMPILER_FEATURE_ENTRY(__cpp_cxx23_stub) ///< Populate eventually
291 };
292 static CompilerFeature cxx23lib[] = {
293     COMPILER_FEATURE_ENTRY(__cpp_lib_cxx23_stub) ///< Populate eventually
294 };
295
296 static CompilerFeature attributes[] = {
297     COMPILER_ATTRIBUTE_ENTRY(carries_dependency)
298     COMPILER_ATTRIBUTE_ENTRY(deprecated)
299     COMPILER_ATTRIBUTE_ENTRY(fallthrough)
300     COMPILER_ATTRIBUTE_ENTRY(likely)
301     COMPILER_ATTRIBUTE_ENTRY(maybe_unused)
302     COMPILER_ATTRIBUTE_ENTRY(nodiscard)
303     COMPILER_ATTRIBUTE_ENTRY(noreturn)
304     COMPILER_ATTRIBUTE_ENTRY(no_unique_address)
305     COMPILER_ATTRIBUTE_ENTRY(unlikely)
306 };
307
308 constexpr bool is_feature_supported(const CompilerFeature& x) {
309     return x.value[0] != '_' && x.value[0] != '0' ;
310 }
311
312 inline void print_compiler_feature(const CompilerFeature& x) {
313     constexpr static int max_name_length = 44; ///< Update if necessary
314     std::string value{ is_feature_supported(x) ? x.value : "-----" };
315     if (value.back() == 'L') value.pop_back(); //~ 201603L -> 201603
316     // value.insert(4, 1, '-'); //~ 201603 -> 2016-03
317     if ( (print.supported_features && is_feature_supported(x))
318         || (print.unsupported_features && !is_feature_supported(x)) ) {
319         std::cout << std::left << std::setw(max_name_length)
320             << x.name << " " << value << '\n';
321     }
322 }
323

```

```
324 template<size_t N>
325 inline void show(char const* title, CompilerFeature (&features)[N]) {
326     if (print.titles) {
327         std::cout << '\n' << std::left << title << '\n';
328     }
329     if (print.sorted_by_value) {
330         std::sort(std::begin(features), std::end(features),
331             [](CompilerFeature const& lhs, CompilerFeature const& rhs) {
332                 return std::strcmp(lhs.value, rhs.value) < 0;
333             });
334     }
335     for (const CompilerFeature& x : features) {
336         print_compiler_feature(x);
337     }
338 }
339
340 int main() {
341     if (print.general_features) show("C++ GENERAL", cxx);
342     if (print.cxx11 && print.core_features) show("C++11 CORE", cxx11);
343     if (print.cxx14 && print.core_features) show("C++14 CORE", cxx14);
344     if (print.cxx14 && print.lib_features ) show("C++14 LIB" , cxx14lib);
345     if (print.cxx17 && print.core_features) show("C++17 CORE", cxx17);
346     if (print.cxx17 && print.lib_features ) show("C++17 LIB" , cxx17lib);
347     if (print.cxx20 && print.core_features) show("C++20 CORE", cxx20);
348     if (print.cxx20 && print.lib_features ) show("C++20 LIB" , cxx20lib);
349     if (print.cxx23 && print.core_features) show("C++23 CORE", cxx23);
350     if (print.cxx23 && print.lib_features ) show("C++23 LIB" , cxx23lib);
351     if (print.attributes) show("ATTRIBUTES", attributes);
352 }
```

---

Длина этого файла очень велика. Если вы хотите узнать больше о каждом макросе, то посетите страницу по тестированию функциональностей языка C++<sup>1</sup>. В частности, эта страница предоставляет ссылку для каждого макроса, так что вы можете получить информацию о каждой функциональности языка. Например, ниже приведена таблица по атрибутам:

---

<sup>1</sup> [https://en.cppreference.com/w/cpp/feature\\_test](https://en.cppreference.com/w/cpp/feature_test).

| <i>attribute-token</i> ⇅ | <b>Attribute</b> ⇅     | <b>Value</b> ⇅ | <b>Standard</b> ⇅ |
|--------------------------|------------------------|----------------|-------------------|
| carries_dependency       | [[carries_dependency]] | 200809L        | (C++11)           |
| deprecated               | [[deprecated]]         | 201309L        | (C++14)           |
| fallthrough              | [[fallthrough]]        | 201603L        | (C++17)           |
| likely                   | [[likely]]             | 201803L        | (C++20)           |
| maybe_unused             | [[maybe_unused]]       | 201603L        | (C++17)           |
| no_unique_address        | [[no_unique_address]]  | 201803L        | (C++20)           |
| nodiscard                | [[nodiscard]]          | 201603L        | (C++17)           |
|                          |                        | 201907L        | (C++20)           |
| noreturn                 | [[noreturn]]           | 200809L        | (C++11)           |
| unlikely                 | [[unlikely]]           | 201803L        | (C++20)           |

Макросы для атрибутов

Демонстрация заголовочного файла <version> и его макросов. Я откомпилировал эту программу на свежих версиях компиляторов GCC, Clang и MSVC. Для GCC и Clang я использовал Compiler Explorer. Флаг /Zc:\_\_cplusplus позволяет получать сообщения от макроса \_\_cplusplus о поддержке новых стандартов C++. Кроме того, я включил поддержку C++20 на всех трех платформах. По очевидным причинам я покажу только поддержку самого языка C++20.

### ○ GCC 10.2

```

C++20 CORE
__cpp_aggregate_paren_init          201902
__cpp_char8_t                      201811
__cpp_concepts                     201907
__cpp_conditional_explicit          201806
__cpp_consteval                    -----
__cpp_constexpr                    201907
__cpp_constexpr_dynamic_alloc       201907
__cpp_constexpr_in_decltype         201711
__cpp_constinit                    201907
__cpp_deduction_guides              201907
__cpp_designated_initializers       201707
__cpp_generic_lambdas               201707
__cpp_impl_coroutine                -----
__cpp_impl_destroying_delete        201806
__cpp_impl_three_way_comparison     201907
__cpp_init_captures                 201803
__cpp_modules                      -----
__cpp_nontype_template_args         201411
__cpp_using_enum                    -----

```

Поддержка возможностей языка C++20 компилятором GCC

## ○ Clang 11.0

|                                 |        |
|---------------------------------|--------|
| C++20 CORE                      |        |
| __cpp_aggregate_paren_init      | -----  |
| __cpp_char8_t                   | 201811 |
| __cpp_concepts                  | 201907 |
| __cpp_conditional_explicit      | 201806 |
| __cpp_consteval                 | -----  |
| __cpp_constexpr                 | 201907 |
| __cpp_constexpr_dynamic_alloc   | 201907 |
| __cpp_constexpr_in_decltype     | 201711 |
| __cpp_constinit                 | 201907 |
| __cpp_deduction_guides          | 201703 |
| __cpp_designated_initializers   | 201707 |
| __cpp_generic_lambdas           | 201707 |
| __cpp_impl_coroutine            | -----  |
| __cpp_impl_destroying_delete    | 201806 |
| __cpp_impl_three_way_comparison | 201907 |
| __cpp_init_captures             | 201803 |
| __cpp_modules                   | -----  |
| __cpp_nontype_template_args     | 201411 |
| __cpp_using_enum                | -----  |

Поддержка возможностей языка C++20 компилятором Clang



## ○ MSVC 19.27

```

C++20 CORE
__cpp_aggregate_paren_init          -----
__cpp_char8_t                      201811
__cpp_concepts                      201811
__cpp_conditional_explicit          201806
__cpp_consteval                    -----
__cpp_constexpr                     201603
__cpp_constexpr_dynamic_alloc       -----
__cpp_constexpr_in_decltype         -----
__cpp_constinit                    -----
__cpp_deduction_guides              201907
__cpp_designated_initializers        201707
__cpp_generic_lambdas               201707
__cpp_impl_coroutine                -----
__cpp_impl_destroying_delete         201806
__cpp_impl_three_way_comparison      201907
__cpp_init_captures                 201803
__cpp_modules                       -----
__cpp_nontype_template_args          201911
__cpp_using_enum                     201907

```

Поддержка возможностей языка C++20 компилятором MSVC

Эти три скриншота несут в себе следующее сообщение: стандарт языка C++20 в конце 2020 года поддерживается всеми тремя компиляторами вполне удовлетворительно.

# 10. Глоссарий

Этот глоссарий ни в коем случае не является исчерпывающим, но он предоставляет ссылки для важных понятий.

## 10.1 Вызываемый объект, Callable

См. Вызываемый объект, Callable Unit

## 10.2 Вызываемый объект, Callable Unit

Вызываемый объект – это что-то, что ведет себя как функция. К вызываемым объектам относятся не только именованные функции, но и функциональные объекты или лямбда-выражения. Если вызываемый объект принимает один аргумент, то он называется унарным вызываемым объектом (unary callable), а с двумя аргументами – бинарным вызываемым объектом (binary callable).

Предикат – это специальный вызываемый объект, который возвращает в качестве результата логическое значение.

## 10.3 Одновременность, concurrency

Одновременность означает, что время выполнения нескольких задач перекрывается. Одновременность – это надмножество параллелизма.

## 10.4 Критическая секция

Критическая секция – это раздел кода, который содержит совместно используемые переменные, которые должны быть защищены для борьбы с состоянием гонки (race condition).

## 10.5 Конфликт по данным, Data race

Конфликт (гонка) по данным – это ситуация, когда два потока одновременно обращаются к совместно используемой переменной. Как минимум один поток пытается изменить переменную, а другой пытается прочесть или также изменить ее. Если в вашей программе есть гонка по данным, то она содержит неопределенное поведение. Это значит, что возможны различные результаты выполнения программы.

## 10.6 Тупик, deadlock

Тупик – это состояние, в котором как минимум один поток навсегда заблокирован, поскольку он ожидает освобождения ресурса, который никогда не получит.

Есть две основные причины тупиков:

- 1) мьютекс не был разблокирован;
- 2) мьютексы запираются в неверном порядке.

## 10.7 Жадное выполнение, eager evaluation

В случае жадного выполнения выражение вычисляется сразу же. Эта стратегия противоположна отложенному выполнению.

## 10.8 Исполнитель, executor

Исполнитель – это объект, связанный с определенным контекстом исполнения. Он предоставляет одну или несколько исполнительных функций для создания агентов исполнения из вызываемого объекта.

## 10.9 Функциональные объекты, function objects

Их не следует называть функторами<sup>1</sup>. Это вполне определенный термин в разделе математики, называемый теорией категорий<sup>2</sup>.

Функциональные объекты – это объекты, которые ведут себя как функции. Они реализуют это за счет выполнения оператора вызова. Функциональные объекты – это объекты, которые могут иметь атрибуты и состояние.

```
struct Square{
    void operator()(int& i){i= i*i;}
};
```

```
std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

```
std::for_each(myVec.begin(), myVec.end(), Square());
```

```
for (auto v: myVec) std::cout << v << " "; // 1 4 9 16 25 36 49 64 81 100
```



### Инстанцируйте функциональные объекты для использования

Распространенной ошибкой является использование имени функционального объекта (Square) в различных алгоритмах вместо экземпляра функционального объекта (Square()): `std::for_each(myVec.begin(), myVec.end(), Square)`. Это типичная ошибка. Нужно использовать экземпляр функционального объекта: `std::for_each(myVec.begin(), myVec.end(), Square())`.

<sup>1</sup> <https://en.wikipedia.org/wiki/Functor>.

<sup>2</sup> [https://en.wikipedia.org/wiki/Category\\_theory](https://en.wikipedia.org/wiki/Category_theory).

## 10.10 Лямбда-выражение, лямбда, lambda

Лямбда реализует свою функциональность прямо на месте. При этом компилятор получает всю необходимую информацию для оптимизации кода. Лямбды могут получать свои аргументы по значению и по ссылке. Они также могут захватывать переменные из своего окружения по значению или по ссылке.

```
std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::for_each(myVec.begin(), myVec.end(), [](int& i){ i= i*i; });
// 1 4 9 16 25 36 49 64 81 100
```

## 10.11 Отложенное (ленивое) выполнение, lazy evaluation

В случае отложенного выполнения<sup>1</sup> выражение выполняется, только если это потребуется. Эта стратегия выполнения противоположна жадному выполнению. Отложенное выполнение часто называют «выполни-если-потребуется».

## 10.12 Отсутствие блокировки, lock-free

Неблокирующий алгоритм свободен от блокировки на уровне системы.

## 10.13 Потерянное пробуждение, lost wakeups

Потерянное пробуждение – это ситуация, в которой поток пропустил свое пробуждение из-за состояния гонки.

## 10.14 Математические законы

Бинарная операция (\*) на некотором множестве X является:

- **ассоциативной**, если она удовлетворяет закону ассоциативности, т. е. для всех  $x, y, z$  из  $X$  справедливо  $(x*y)*z=x*(y*z)$ ;
- **коммутативной**, если она удовлетворяет закону коммутативности, т. е. для всех  $x, y$  из  $X$  справедливо  $x*y=y*x$ ;
- **дистрибутивной**, если она удовлетворяет дистрибутивному закону, т. е. для всех  $x, y, z$  из  $X$  справедливо  $x(y+z)=xy+xz$ .

## 10.15 Расположение в памяти, memory location

Расположением в памяти в соответствии с [cppreference.com](http://cppreference.com) является:

- объект скалярного типа (арифметический тип, тип указателя, перечисления или `std::nullptr_t`);
- наибольшая непрерывная последовательность битовых полей ненулевой длины.

<sup>1</sup> [https://en.wikipedia.org/wiki/Lazy\\_evaluation](https://en.wikipedia.org/wiki/Lazy_evaluation).

## 10.16 Модель памяти, memory model

Модель памяти определяет взаимоотношения между объектами и расположениями в памяти и отвечает на вопрос: что произойдет, если два потока обратятся к одним и тем же участкам в памяти?

## 10.17 Неблокирующий, non-blocking

Алгоритм называется неблокирующим, если ошибки или приостановка одного потока не могут вызывать ошибки или приостановки другого потока. Это определение взято из отличной книги *Java concurrency in practice*<sup>1</sup>.

## 10.18 Объект, object

Тип является объектом, если это скаляр, массив, объединение или класс.

## 10.19 Параллелизм, parallelism

Параллелизм означает, что несколько задач выполняются в одно и то же время. Параллелизм является подмножеством одновременности (concurrency). В отличие от одновременности, параллелизм требует, чтобы программа выполнялась на нескольких ядрах.

## 10.20 Предикат, predicate

Предикаты – это вызываемые блоки, которые возвращают логические значения. Если предикат принимает один аргумент, то он называется унарным. Если у предиката два аргумента, то он называется бинарным.

## 10.21 RAII

Аббревиатура RAII (Resource Acquisition Is Initialization) обозначает популярную технику в C++, при которой получение и освобождение ресурса связано со временем жизни объекта. Для блокировки (lock) это значит, что мьютекс будет заблокирован в конструкторе и разблокирован в деструкторе.

Типичными примерами использования в C++ являются различные блокировки (locks), которые запирают мьютекс, умные указатели, которые отвечают за время жизни ресурса (памяти), или контейнеры из стандартной библиотеки шаблонов<sup>2</sup>, которые управляют временем жизни содержащихся в них объектов.

## 10.22 Состояние гонки, race conditions

Состояние гонки – это ситуация, в которой результат операции зависит от упорядочения отдельных операций.

<sup>1</sup> <http://jcip.net/>.

<sup>2</sup> <https://en.cppreference.com/w/cpp/container>.

Состояние гонки очень тяжело обнаружить. Происходят они или нет, зависит от «переплетения потоков». Это значит, что количество ядер, использование системы или уровень оптимизации исполняемого файла могут быть причинами, из-за которых появилось состояние гонки.

## 10.23 Регулярный, *regular*

В добавление к требованиям концепта *SemiRegular* концепт *Regular* требует, чтобы тип был равносопоставимым.

## 10.24 Скалярный, *scalar*

Скалярный тип – это либо арифметический тип (`std::is_arithmetic`<sup>1</sup>), либо перечисление (`enum`), указатель, указатель на поле объекта или `std::nullptr_t`.

## 10.25 Полурегулярный, *SemiRegular*

Полурегулярный тип *X* должен поддерживать Большую шестерку и операцию `swap(X&, X&)`.

## 10.26 Ложное пробуждение, *Spurious Wakeup*

Ложное пробуждение – это ошибочное уведомление. Ожидающая часть условной переменной или атомарного флага может получить уведомление, хотя соответствующее уведомление не посылалось.

## 10.27 Большая четверка, *Big Four*

Большая четверка – это четыре главные новые возможности C++20: концепты, модули, библиотека диапазонов и сопрограммы.

- **Концепты** меняют то, как мы думаем о программировании с использованием шаблонов. Они являются семантическими категориями для параметров шаблонов. Они позволяют выражать явно намерения при помощи системы типов. Если что-то идет не так, то компилятор выдаст понятное сообщение об ошибке.
- **Модули** позволяют преодолеть ограничения заголовочных файлов. Они очень многообещающи. Например, разделение заголовочных и исходных файлов становится устаревшим, как и сам препроцессор. В результате это приводит к более быстрой сборке и более удобному способу создания пакетов.
- **Новая библиотека диапазонов** поддерживает выполнение алгоритмов непосредственно над контейнерами, комбинирование алгоритмов при помощи символа `|` и отложенное применение алгоритмов к бесконечным потокам данных.

<sup>1</sup> [https://en.cppreference.com/w/cpp/types/is\\_arithmetic](https://en.cppreference.com/w/cpp/types/is_arithmetic).

- Благодаря **сопрограммам** асинхронное программирование в C++ становится распространенным. Сопрограммы являются базисом для взаимодействующих задач, циклов обработки событий, бесконечных потоков данных и конвейеров.

## 10.28 Большая шестерка, Big Six

Большая шестерка состоит из следующих функций:

- конструктор по умолчанию: `X()`;
- конструктор копирования `X(const X&)`;
- операция копирования `X& operator = ( const X& )`;
- конструктор перемещения `X(&&)`;
- присвоение перемещением `X& operator = (X&&)`;
- деструктор `~X()`.

## 10.29 Поток, Thread

Поток выполнения – это мельчайшая последовательность команд, которой планировщик может управлять независимо. Планировщик обычно является частью операционной системы. Реализация потоков и процессов может отличаться для различных операционных систем, но в большинстве случаев поток – это компонент процесса. Внутри одного процесса может существовать несколько потоков, выполняющихся параллельно и совместно, а также использующих такие ресурсы, как память, в то время как различные процессы не могут совместно использовать эти ресурсы. За подробностями вы можете обратиться к статье в Википедии о потоках<sup>1</sup>.

## 10.30 Сложность по времени, Time Complexity

$O(i)$  обозначает сложность по времени (времени выполнения) какой-либо операции. Запись  $O(1)$  означает, что время выполнения операции над контейнером является постоянной величиной и поэтому не зависит от его размера. В отличие от этого,  $O(n)$  значит, что время выполнения линейно зависит от количества элементов в контейнере.

## 10.31 Единица трансляции, Translation Unit

Единица трансляции – это исходный файл после выполнения препроцессора языка C. Препроцессор языка C включает заголовочные файлы при помощи директив `#include`, выполняет условное включение при помощи таких директив, как `#ifdef` и `#ifndef`, и выполняет подстановку макросов. Компилятор использует единицы трансляции для создания объектных файлов.

<sup>1</sup> [https://en.wikipedia.org/wiki/Thread\\_\(computing\)](https://en.wikipedia.org/wiki/Thread_(computing)).

## 10.32 Неопределенное поведение, Undefined Behavior

Это значит, что ваша программа может выдать правильный результат, неправильный результат, неожиданно прекратить выполнение или просто не скомпилироваться. Это поведение может измениться при переносе на новую платформу, переходе к новому компилятору или в результате никак не связанного с этим изменения кода.



Книги издательства «ДМК Пресс» можно заказать  
в торгово-издательском холдинге «КТК Галактика» наложенным платежом,  
выслав открытку или письмо по почтовому адресу:  
115487, г. Москва, пр. Андропова д. 38 оф. 10.  
При оформлении заказа следует указать адрес (полностью),  
по которому должны быть высланы книги;  
фамилию, имя и отчество получателя.  
Желательно также указать свой телефон и электронный адрес.  
Эти книги вы можете заказать и в интернет-магазине: [www.galaktika-dmk.com](http://www.galaktika-dmk.com).  
Оптовые закупки: тел. (499) 782-38-89.  
Электронный адрес: [books@aliants-kniga.ru](mailto:books@aliants-kniga.ru).

**Райнер Гримм**

**С++20 в деталях**

|                         |                                                            |
|-------------------------|------------------------------------------------------------|
| Главный редактор        | <i>Мовчан Д. А.</i>                                        |
|                         | <a href="mailto:dmkpress@gmail.com">dmkpress@gmail.com</a> |
| Зам. главного редактора | <i>Сенченкова Е. А.</i>                                    |
| Перевод                 | <i>Боресков А. В.</i>                                      |
| Научные редакторы       | <i>Романов А. Ю., Романова И. И.</i>                       |
| Корректор               | <i>Синяева Г. И.</i>                                       |
| Верстка                 | <i>Луценко С. В.</i>                                       |
| Дизайн обложки          | <i>Мовчан А. Г.</i>                                        |

Формат 70×100 1/16.

Гарнитура «PT Serif». Печать цифровая.

Усл. печ. л. 42,09. Тираж 200 экз.

Веб-сайт издательства: [www.dmkpress.com](http://www.dmkpress.com)