

Updated for C++ 20



A Tour of C++

Third Edition

СБЕР ЧАЕВЫЕ



Поддержи переводчика
ВЯЧЕСЛАВ КИРЮХИН

Bjarne Stroustrup



C++ In-Depth Series Bjarne Stroustrup

Содержание

Содержание.....	2
Тур по C++	9
Благодарность	11
Основы.....	12
1.1 Введение.....	12
1.2 Программы.....	12
1.2.1 Hello, World!	13
1.3 Функции.....	15
1.4 Типы, Переменные и Арифметика.....	16
1.4.1 Арифметика.....	17
1.4.2 Инициализация.....	19
1.5 Область видимости и время жизни.....	20
1.6 Константы	21
1.7 Указатели, Массивы и Ссылки	22
1.7.1 Нулевой указатель	24
1.8 Условные операторы	25
1.9 Сопоставление с аппаратным обеспечением	27
1.9.1 Присваивание	27
1.9.2 Инициализация.....	28
1.10 Советы.....	29
Пользовательские типы	31
2.1 Введение.....	31
2.2 Структуры.....	32
2.3 Классы.....	33
2.4 Перечисления	35
2.5 Объединения.....	36
2.6 Советы	38
Модульность.....	39
3.1 Введение.....	39
3.2 Раздельная компиляция	40
3.2.1 Заголовочные файлы.....	40
3.2.2 Модули	42
3.3 Пространства имён	45
3.4 Аргументы функции и возвращаемые значения	47
3.4.1 Передача аргументов.....	47
3.4.2 Возвращение значений.....	48
3.4.3 Выведение типа возвращаемого значения	50
3.4.4 Суффиксная запись типа возвращаемого значения.....	50
3.4.5 Структурное связывание.....	50
3.5 Советы	52
Обработка ошибок.....	53
4.1 Введение.....	53
4.2 Исключения	54
4.3 Инварианты.....	55
4.4 Альтернативные способы обработки ошибок	57

4.5 Утверждения.....	59
4.5.1 assert().....	60
4.5.2 static_assert().....	60
4.5.3 noexcept.....	61
4.6 Советы.....	61
Классы.....	63
5.1 Введение.....	63
5.1.1 Классы.....	64
5.2 Конкретные типы.....	64
5.2.1 Арифметические типы.....	65
5.2.2 Контейнеры.....	67
5.2.3 Инициализация контейнеров.....	68
5.3 Абстрактные типы.....	70
5.4 Виртуальные функции.....	73
5.5 Иерархии классов.....	73
5.5.1 Преимущества иерархий.....	76
5.5.2 Навигация в иерархии.....	77
5.5.3 Предотвращение утечки ресурсов.....	78
5.6 Советы.....	79
Основные операции.....	81
6.1 Введение.....	81
6.1.1 Основные операции.....	81
6.1.2 Преобразования типов.....	83
6.1.3 Инициализация элементов.....	84
6.2 Копирование и перемещение.....	84
6.2.1 Копирование контейнеров.....	84
6.2.2 Перемещение контейнеров.....	86
6.3 Управление ресурсами.....	88
6.4 Перегрузка операторов.....	90
6.5 Стандартные операции.....	91
6.5.1 Операторы сравнения.....	91
6.5.2 Операции с контейнерами.....	92
6.5.3 Итераторы и “умные указатели”.....	93
6.5.4 Операции ввода-вывода.....	93
6.5.5 swap().....	94
6.5.6 hash<>.....	94
6.6 Пользовательские литералы.....	94
6.7 Советы.....	95
Шаблоны.....	97
7.1 Введение.....	97
7.2 Параметризованные типы.....	97
7.2.1 Ограниченные аргументы шаблона.....	99
7.2.2 Аргументы-значения шаблона.....	100
7.2.3 Выведение типов аргументов шаблонов.....	101
7.3 Параметризованные операции.....	102
7.3.1 Шаблоны функций.....	103
7.3.2 Функциональные объекты.....	103

7.3.3 Лямбда выражения.....	105
7.3.3.1 Лямбды как аргументы функции	105
7.3.3.2 Лямбды для инициализации	106
7.3.3.3 Напоследок, finally()	108
7.4 Механизмы шаблонов	108
7.4.1 Шаблоны переменных	109
7.4.2 Псевдонимы.....	109
7.4.3 if времени компиляции.....	110
7.5 Советы	111
Концепты и обобщенное программирование	113
8.1 Введение.....	113
8.2 Концепты	114
8.2.1 Использование концептов	114
8.2.2 Перегрузка основанная на концептах	116
8.2.3 Правильный код.....	117
8.2.4 Определение концептов	117
8.2.4.1 Проверка определения	120
8.2.5 Концепты и auto	120
8.2.6 Концепты и типы.....	121
8.3 Обобщённое программирование	122
8.3.1 Использование концептов	122
8.3.2 Абстракции использующие шаблоны.....	123
8.4 Шаблоны с переменным числом аргументов	124
8.4.1 Выражения свёртки.....	125
8.4.2 Передача аргументов.....	126
8.5 Модель компиляции шаблонов.....	127
8.6 Советы	128
Обзор стандартной библиотеки	129
9.1 Введение.....	129
9.2 Компоненты стандартной библиотеки	130
9.3 Организация стандартной библиотеки	131
9.3.1 Пространства имён	131
9.3.2 Пространство имён ranges	132
9.3.3 Модули	133
9.3.4 Заголовочные файлы.....	133
9.4 Советы	134
Строки и регулярные выражения.....	135
10.1 Введение	135
10.2 Строки	135
10.2.1 Реализация string.....	137
10.3 Строковые представления	138
10.4 Регулярные выражения	139
10.4.1 Поиск	140
10.4.2 Описание регулярных выражений	141
10.4.3 Итераторы	145
10.5 Советы.....	145

Ввод и вывод.....	148
11.1 Введение.....	148
11.2 Вывод.....	149
11.3 Ввод.....	150
11.4 Состояния потоков I/O.....	151
11.5 I/O пользовательских типов.....	152
11.6 Форматирование вывода.....	153
11.6.1 Форматирование потока	153
11.6.2 Форматирование в стиле printf().....	155
11.7 Потоки	157
11.7.1 Стандартные потоки.....	157
11.7.2 Файловые потоки	158
11.7.3 Строковые потоки.....	158
11.7.4 Потоки памяти	159
11.7.5 Синхронизированные потоки.....	159
11.8 I/O в стиле Си.....	160
11.9 Файловая система.....	160
11.9.1 Пути	161
11.9.2 Файлы и каталоги	163
11.10 Советы.....	164
Контейнеры.....	166
12.1 Введение.....	166
12.2 vector	166
12.2.1 Элементы	169
12.2.2 Проверка диапазона.....	169
12.3 list	171
12.5 map.....	173
12.6 unordered_map.....	174
12.7 Аллокаторы	175
12.8 Обзор контейнеров.....	177
12.9 Советы	178
Алгоритмы.....	181
13.1 Введение.....	181
13.2 Применение итераторов.....	183
13.3 Типы итераторов	185
13.3.1 Поточковые итераторы	186
13.4 Использование предикатов.....	188
13.5 Обзор алгоритмов	189
13.6 Параллельные алгоритмы	190
13.7 Советы	191
Диапазоны.....	192
14.1 Введение.....	192
14.2 Представления	193
14.3 Генераторы	195
14.4 Конвейеры.....	195
14.5 Обзор концептов	196
14.5.1 Концепты типов.....	197

14.5.2 Концепты итераторов	199
14.5.3 Концепты диапазонов	200
14.6 Советы	201
Умные указатели и контейнеры	202
15.1 Введение	202
15.2 Указатели	203
15.2.1 unique_ptr и shared_ptr	204
15.2.2 span	207
15.3 Контейнеры	208
15.3.1 array	209
15.3.2 bitset	211
15.3.3 pair	212
15.3.4 tuple	214
15.4 Альтернативы	215
15.4.1 variant	215
15.4.2 optional	217
15.4.3 any	218
15.5 Советы	218
Утилиты	220
16.1 Введение	220
16.2 Время	220
16.2.1 Часы	221
16.2.2 Календари	221
16.2.3 Временные зоны	222
16.3 Адаптация функций	223
16.3.1 Лямбды как адапторы	223
16.3.2 mem_fn()	223
16.3.3 function	224
16.4 Функция типа	224
16.4.1 Предикаты типа	225
16.4.2 Условные свойства	227
16.4.3 Генераторы типов	228
16.4.4 Связанные типы	228
16.5 source_location	229
16.6 move() and forward()	229
16.7 Битовые манипуляции	231
16.8 Выход из программы	232
16.9 Советы	232
Числовые вычисления	234
17.1 Введение	234
17.2 Математические функции	235
17.3 Численные алгоритмы	236
17.3.1 Многопоточные численные алгоритмы	236
17.4 Комплексные числа	237
17.5 Случайные числа	238
17.6 Векторная арифметика	240
17.7 Числовые ограничения	240

17.8 Псевдонимы типов.....	240
17.9 Математические константы	241
17.10 Советы.....	241
Параллелизм.....	243
18.1 Введение.....	243
18.2 Задачи и thread.....	244
18.2.1 Передача аргументов	245
18.2.2 Возвращение результатов	246
18.3 Обмен данными	247
18.3.1 mutex и блокировки	247
18.3.2 atomic	248
18.4 Ожидание событий	249
18.5 Коммуникации задач	250
18.5.1 future и promise	251
18.5.2 packaged_task	252
18.5.3 async().....	253
18.5.4 Остановка thread.....	254
18.6 Корутин (сопрограммы).....	255
18.6.1 Кооперативная многозадачность	256
18.7 Советы	259
История и совместимость	261
19.1 История	261
19.1.1 Временная шкала.....	262
19.1.2 Ранние годы	263
19.1.3 Стандарты ISO C++	266
19.1.4 Стандарты и стиль.....	268
19.1.5 Использование C++	269
19.1.6 Модель C++	269
19.2 Эволюция функций C++	270
19.2.1 Языковые особенности C++11	270
19.2.2 Языковые особенности C++14	271
19.2.3 Языковые особенности C++17	271
19.2.4 Языковые особенности C++20	272
19.2.5 Компоненты стандартной библиотеки C++11	272
19.2.6 Компоненты стандартной библиотеки C++14	273
19.2.7 Компоненты стандартной библиотеки C++17	273
19.2.8 Компоненты стандартной библиотеки C++20	273
19.2.9 Удаленные и устаревшие функции	274
19.3 Совместимость C/C++	275
19.3.1 C и C++ - родные братья	275
19.3.2 Проблемы совместимости	277
19.3.2.1 Проблемы стиля	277
19.3.2.2 void*	278
19.3.2.3 Линковка	279
19.4 Библиография	279
19.5 Советы	282

Модуль std.....	284
A.1 Введение	284
A.2 Используйте то, что предлагает Ваша реализация	285
A.3 Используйте заголовки	285
A.4 Сделайте свой собственный module std	285
A.5 Советы	286
Index	287

Тур по С++

Третье издание

Bjarne Stroustrup

Coverphoto by: Marco Pagnolato ([Unsplash.com](https://unsplash.com/@marco_pagnolato): @marco_pagnolato).

Author photo courtesy of Bjarne Stroustrup.

This book was typeset in Times and Helvetica by the author.

ISBN-13: 978-0-13-681648-5

ISBN-10: 0-13-681648-7

First printing, October 2022

Предисловие

*Когда вы захотите проинструктировать,
будьте кратки.
– Цицерон*

C++ ощущается как новый язык. Сегодня я могу выражать свои идеи чётче, проще и непосредственнее, чем в C++98 или C++11. Кроме того, в результате программы лучше проверяются компилятором и быстрее работают.

В этой книге дается обзор C++, как он определен в C++20, текущем стандарте ISO C++, и реализован основными поставщиками C++. Кроме того, в ней упоминается пара библиотечных компонентов, используемых в настоящее время, но не планируемых к включению в стандарт до C++23.

Как и другие современные языки, C++ велик, и для эффективного использования требуется большое количество библиотек. Цель этой небольшой книги - дать опытному программисту представление о том, что представляет собой современный C++. Книга охватывает большинство основных языковых функций и основные компоненты стандартной библиотеки. Эту книгу можно прочитать всего за день или два, но, очевидно, что требуется гораздо больше времени, чтобы научиться писать хороший код на C++. Однако цель здесь не в овладении мастерством, а в том, чтобы дать обзор языка, привести ключевые примеры и помочь программисту начать работу.

Предполагается, что вы уже программировали раньше. Если нет, пожалуйста, подумайте о прочтении учебника например *Программирование: принципы и практика использования C++ (Второе издание)* [[Stroustrup, 2014](#)], прежде чем продолжать чтение этой книги. Даже если вы программировали раньше, язык, который вы использовали, или приложения, которые вы писали, могут сильно отличаться от стиля C++, представленного здесь.

Представьте себе обзорную экскурсию по какому-нибудь городу, например, Копенгагену или Нью-Йорку. Всего за несколько часов вам покажут основные достопримечательности, расскажут несколько историй и дадут несколько советов о том, что делать дальше. После такой экскурсии вы *не* знаете город. Вы *не* понимаете всего, что видели и слышали; некоторые истории могут показаться странными или даже неправдоподобными. Вы *не* знаете, как ориентироваться в формальных и неформальных правилах, которые управляют жизнью в городе. Чтобы по-настоящему узнать город, нужно пожить в нем, часто несколько лет. Однако, если вам немного повезет, вы получите общее представление о том, что особенного в городе, и идеи о том, что может вас заинтересовать. После экскурсии может начаться настоящее исследование города.

В этой экскурсии представлены основные возможности языка C++, поддерживаемые ими стили программирования, такие как объектно-ориентированное и обобщённое программирование. В нем не делается попытка предоставить подробное, похожее на справочник руководство, представление языка по отдельным функциям. В лучших традициях учебников я пытаюсь объяснить ту или иную функцию перед ее использованием, но это не всегда возможно, и не все читают текст строго последовательно. Я предполагаю некоторую техническую зрелость от моих читателей. Итак, читателю рекомендуется использовать перекрестные ссылки и алфавитный указатель.

Аналогично, стандартные библиотеки представлены в виде примеров, а не в виде полного описания. Читателю рекомендуется искать дополнительные и вспомогательные материалы по мере необходимости. Экосистема C++ - это гораздо больше, чем просто возможности, предлагаемые стандартом ISO (например, библиотеки, системы сборки, инструменты анализа и среды разработки). В Интернете доступно огромное количество материалов (разного качества). Большинство читателей найдут полезные обучающие и обзорные видеоролики с конференций CppCon и Meeting C++. Для получения технических подробностей о языке и библиотеке, предлагаемых стандартом ISO C++, я рекомендую [\[Cppreference\]](#). Например, когда я упоминаю функцию или класс стандартной библиотеки, можно легко найти их определение в других источниках, а изучив документацию по ним, можно найти множество связанных с ними средств.

В этом туре C++ представлен как единое целое, а не как слоеный пирог. Следовательно, я редко идентифицирую языковые функции как присутствующие в C, C++98 или более поздних стандартах ISO. Такую информацию можно найти в [главе 19](#) (История и совместимость). Я сосредотачиваюсь на основах и стараюсь быть кратким, но я не полностью устоял перед искушением представить новые функции, такие как модули ([§3.2.2](#)), концепты ([§8.2](#)) и корутины ([§18.6](#)). Небольшое предпочтение свежим разработкам также, по-видимому, удовлетворяет любопытство многих читателей, которые уже знакомы с какой-либо более старой версией C++.

Справочное руководство по языку программирования или стандарт просто указывают, что можно сделать с помощью языка, но программисты часто больше заинтересованы в том, чтобы научиться хорошо использовать язык программирования. Этот аспект частично рассматривается в ряде затронутых тем, частично в тексте и, в частности, в разделах с рекомендациями. Дополнительные рекомендации о том, что собой представляет хороший современный C++, можно найти в руководстве C++ Core Guidelines [\[Stroustrup, 2015\]](#). Core Guidelines могут стать хорошим источником для дальнейшего изучения идей, представленных в этой книге. Вы можете отметить поразительное сходство формулировок рекомендаций и даже нумерации рекомендаций между Core Guidelines и этой книгой. Одна из причин заключается в том, что первое издание A Tour of C++ было основным источником первоначальных Core Guidelines.

Благодарность

Спасибо всем, кто помогал дополнять и исправлять предыдущие выпуски “Эксперсии по C++”, особенно студентам моего курса “Проектирование с использованием C++” в Колумбийском университете. Спасибо Morgan Stanley за то, что дали мне время написать это третье издание. Спасибо Чаку Эллисону, Гаю Дэвидсону, Стивену Дьюхерсту, Кейт Грегори, Дэнни Калеву, Гору Нишанову и Дж.Си ван Винкелю за рецензию на книгу и предложения по многим улучшениям.

Эта книга была создана автором с использованием troff с использованием макросов Брайана Кернигана.

Manhattan, New York

Bjarne Stroustrup

Основы

*Первое, что мы сделаем, давайте
убьем всех юристов – Генрих VI,
Part II*

- [Введение](#)
- [Программы](#)

[Hello, World!](#)

- [Функции](#)
- [Типы, Переменные и Арифметика](#)

[Арифметика; Инициализация](#)

- [Область видимости и время жизни](#)
- [Константы](#)
- [Указатели, Массивы и Ссылки](#)

[Нулевой указатель](#)

- [Условные операторы](#)
- [Сопоставление с аппаратным обеспечением](#)

[Присвоение; Инициализация](#)

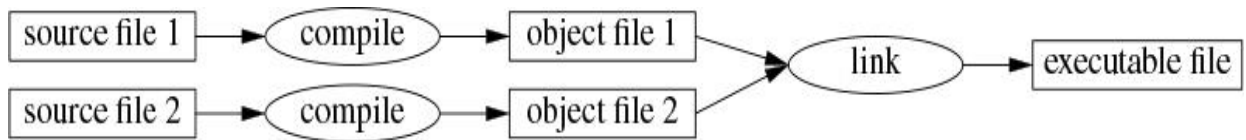
- [Советы](#)

1.1 Введение

В этой главе неофициально представлены обозначения C++, модель памяти и вычислений C++, а также основные механизмы организации кода в программе. Все эти языковые средства, которые наиболее часто встречающиеся в C и иногда называются *процедурным программированием*.

1.2 Программы

C++ - это компилируемый язык. Для запуска программы ее исходный текст должен быть обработан компилятором, создающим объектные файлы, которые объединяются компоновщиком, получая исполняемую программу. Программа на C++ обычно состоит из множества файлов исходного кода (обычно называемых просто *исходными файлами*).



Исполняемая программа создается для определенной комбинации аппаратной платформы и операционной системы; она не переносима, скажем, с устройства Android на персональный компьютер с Windows. Когда мы говорим о переносимости программ на C++, мы обычно имеем в виду переносимость исходного кода; то есть исходный код может быть успешно скомпилирован и запущен в различных системах.

Стандарт ISO C++ определяет два типа сущностей:

- *Фундаментальные возможности языка*, такие как встроенные типы (например, `char` и `int`) и циклы (например, операторы `for` и `while`)
- *Компоненты стандартной библиотеки STL*, такие как контейнеры (например, `vector` и `map`) и операции ввода-вывода (например, `<<` и `getline()`)

Компоненты стандартной библиотеки - это совершенно обычный код на C++, предоставляемый каждой реализацией языка C++. То есть стандартная библиотека C++ может быть реализована в самом C++ и является таковой (с очень незначительным использованием машинного кода для таких вещей, как переключение контекста `thread`). Это означает, что C++ достаточно выразителен и эффективен для самых сложных задач системного программирования.

C++ - это статически типизированный язык. То есть тип каждой сущности (например, объекта, значения, имени и выражения) должен быть известен компилятору в момент его использования. Тип объекта определяет набор операций, применимых к нему, и его расположение в памяти.

1.2.1 Hello, World!

Минимальная программа на C++ выглядит следующим образом:

```
int main() { }           // the minimal C++ program
```

Здесь определена функция с именем `main`, которая не принимает аргументов и ничего не делает.

Фигурные скобки, `{ }`, группируют выражения в C++. Здесь они указывают начало и конец тела функции. Двойная косая черта, `//`, начинает комментарий, который продолжается до конца строки. Комментарий предназначен для обычного чтения человеком, а компилятор полностью игнорирует комментарии.

Каждая программа на C++ должна иметь ровно одну глобальную функцию с именем `main()`. Программа начинается с выполнения этой функции. Целочисленное значение `int`, возвращаемое `main()`, если таковое имеется, является значением, возвращаемым программой "системе". Если значение не будет возвращено, система получит значение, указывающее на успешное завершение. Ненулевое значение возвращаемое `main()` указывает на сбой. Не каждая операционная система и среда выполнения используют это возвращаемое значение: среды на базе Linux/Unix используют, но среды на базе Windows делают это редко.

Как правило, программа выдает некоторый вывод. Вот программа, которая выводит в консоль **Hello, World!**:

```
import std;

int main()
{
```

```
std::cout << "Hello, World!\n";
}
```

Строка `import std;` инструктирует компилятор что необходимо сделать доступными объявления стандартной библиотеки. Без этих объявлений выражение

```
std::cout << "Hello, World!\n"
```

не имело бы никакого смысла. Оператор `<<` (“put to”) записывает свой второй аргумент в свой первый. В данном случае строковый литерал `"Hello, World!\n"` записывается в стандартный поток вывода `std::cout`. Строковый литерал - это последовательность символов, заключенная в двойные кавычки. В строковом литерале символ обратной косой черты `\`, за которым следует другой символ, обозначает один “специальный символ”. В этом случае `\n` - это символ перевода курсора на новую строку, так что написанные символы - это `Hello, World!` за ним следует переход на новую строку.

`std::` указывает, что имя `cout` находится в пространстве имен стандартной библиотеки (§3.3). Я обычно опускаю `std::` при обсуждении стандартных функций; в §3.3 показано, как сделать имена из пространства имен видимыми без явного указания.

Директива `import` является новой в C++20, и представление всей стандартной библиотеки в виде модуля `std` еще не является стандартным. Это будет объяснено в §3.2.2. Если у вас возникли проблемы с `import std;`, попробуйте обычный старомодный

```
#include <iostream>           // include the declarations for the I/O
                               // stream library
int main()
{
    std::cout << "Hello, World!\n";
}
```

Более подробно это будет объяснено в §3.2.1 и работает во всех реализациях C++ с 1998 года (§19.1.1).

По сути, весь исполняемый код помещается в функции и вызывается прямо или косвенно из `main()`. Например:

```
import std;                   // import the declarations for the standart library
using namespace std;         // make names from std visible without std:: (§3.3)

double square(double x) {    //square a double-precision floating-point number
    return x*x;
}

void print_square(double x) {
    cout << "the square of " << x << " is " << square(x) << "\n";
}

int main() {
    print_square(1.234);      // print: the square of 1.234 is 1.52276
}
```

Тип возвращаемого значения `void` указывает на то, что функция не возвращает значение.

1.3 Функции

Основной способ заставить что-то сделать программу на C++ - это вызвать соответствующую функцию. Определение функции - это способ указать, как должны выполняться необходимые действия. Функция не может быть вызвана, если она не была объявлена ранее.

Объявление функции указывает имя функции, тип возвращаемого значения (если таковое имеется), а также количество и типы аргументов, которые должны быть переданы при вызове. Например:

```
Elem* next_elem();           // no argument; return a pointer to Elem (an Elem*)
void exit(int);              // int argument; return nothing
double sqrt(double);         // double argument; return a double
```

В объявлении функции возвращаемый тип указывается перед именем функции, а типы аргументов - после имени, заключенными в круглые скобки.

Семантика передачи аргументов идентична семантике инициализации (§3.4.1). То есть проверяются типы аргументов, и при необходимости выполняется неявное преобразование типов аргументов (§1.4). Например:

```
double s2 = sqrt(2);         // call sqrt() with the argument double{2}
double s3 = sqrt("three");    // error: sqrt() requires an argument of type double
```

Ценность такой проверки во время компиляции и преобразования типов не следует недооценивать.

Объявление функции может содержать имена аргументов. Это может помочь читателю программы, но, если объявление не является также определением функции, компилятор просто игнорирует такие имена. Например:

```
double sqrt(double d);       // return the square root of d
double square(double);       // return the square of the argument
```

Тип функции состоит из типа возвращаемого значения, за которым следует последовательность типов ее аргументов в круглых скобках. Например:

```
double get(const vector<double>& vec, int index); // type: double(const
                                                    // vector<double>&,int)
```

Функция может быть членом класса (§2.3, §5.2.1). Для такой функции-члена имя ее класса также является частью типа функции. Например:

```
char& String::operator[](int index);           // type: char& String::(int)
```

Мы хотим, чтобы наш код был понятным, потому что это первый шаг на пути к удобству его обслуживания. Первым шагом к пониманию является разбиение вычислительных задач на значимые фрагменты (представленные в виде функций и классов) и присвоение им имен. Затем такие функции составляют базовый словарь вычислений, точно так же, как типы (встроенные и определяемые пользователем) составляют базовый словарь данных. Стандартные алгоритмы C++ (например, `find`, `sort` и `iota`) обеспечивают хорошее начало (глава 13). Далее мы можем объединять функции, представляющие общие или специализированные задачи, в более крупные вычисления.

Количество ошибок в коде сильно коррелирует с объемом кода и его сложностью. Обе проблемы можно решить, используя большее количество функций и делая функции короче. Использование функции для выполнения определенной задачи часто избавляет нас от написания определенного фрагмента кода посреди другого кода; пре-

вращение его в функцию заставляет нас называть действие и документировать его зависимости. Если мы не можем найти подходящее название, высока вероятность того, что у нас проблема с дизайном.

Если определены две функции с одинаковым именем, но с разными типами аргументов, компилятор выберет наиболее подходящую функцию для каждого вызова. Например:

```
void print(int);           // takes an integer argument
void print(double);        // takes a floating-point argument
void print(string);        // takes a string argument

void user() {
    print(42);              // calls print(int)
    print(9.65);           // calls print(double)
    print("Barcelona");     // calls print(string)
}
```

Если могут быть вызваны две альтернативные функции, но ни одна из них не подходит лучше другой, вызов считается неоднозначным и компилятор выдает ошибку. Например:

```
void print(int, double);
void print(double, int);

void user2() {
    print(0,0);             // error: ambiguous
}
```

Определение нескольких функций с одинаковым именем называется *перегрузкой функций* и является одной из важнейших частей обобщённого программирования (§8.2). При перегрузке функций, каждая функция с одинаковым именем должна реализовывать одинаковую семантику. Функции `print()` являются примером этого; каждая функция `print()` выводит свой аргумент.

1.4 Типы, Переменные и Арифметика

Каждое имя и каждое выражение имеют тип, который определяет, какие операции могут быть выполнены над ними. Например, объявление

```
int inch;
```

указывает, что `inch` имеет тип `int`; то есть `inch` является целочисленной переменной. Объявление - это инструкция, которая вводит объект в программу и определяет его тип:

- *Тип* определяет набор возможных значений и набор операций (для объекта).
- *Объект* - это некоторая память, которая содержит значение некоторого типа.
- *Значение* - это набор битов, интерпретируемых в соответствии с типом.
- *Переменная* - это именованный объект.

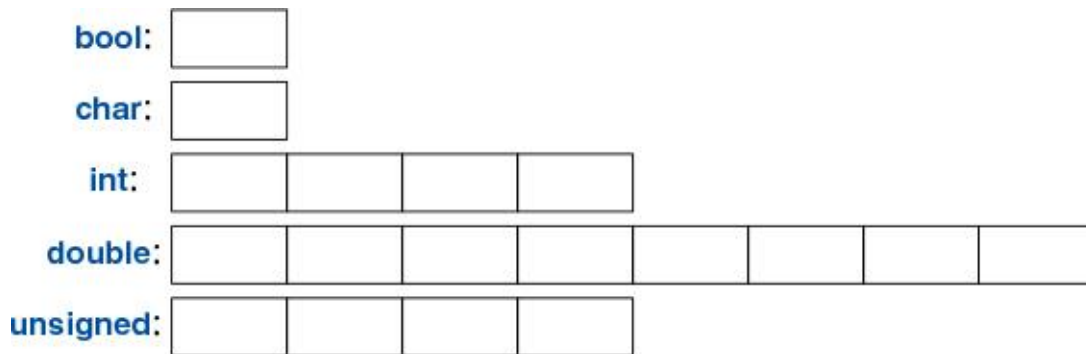
C++ предлагает небольшой зоопарк фундаментальных типов, но поскольку я не зоолог, я не буду перечислять их все. Вы можете найти их в справочниках, например в [\[Cppreference\]](#) в Интернете. Примеры фундаментальных типов:

```

bool           // Boolean, possible values are true and false
char           // character, for example, 'a', 'z', and 9
int            // integer, for example, -273, 42, and 1066
double         // double-precision floating-point number, for example,
               // -273.15, 3.14, and 6.626e-34
unsigned       // non-negative integer, for example, 0, 1, and 999
               // (используется для побитовых логических операций)

```

Каждый базовый тип непосредственно соответствует аппаратным средствам и имеет фиксированный размер, который определяет диапазон значений, которые могут в нем храниться:



Переменная типа `char` имеет естественный размер для хранения символа на данной машине (обычно 8-разрядный байт), а размеры других типов кратны размеру символа `char`. Размер типа определяется реализацией (т.е. он может варьироваться на разных машинах и операционных системах) и может быть получен с помощью оператора `sizeof`; например, `sizeof(char)` равен 1, а `sizeof(int)` часто равен 4. Когда нам нужен тип определенного размера, мы используем псевдоним типа стандартной библиотеки, такой как `int32_t` (§17.8).

Числа могут быть как целыми, так и с плавающей запятой.

- Литералы чисел с плавающей запятой распознаются по десятичной точке (например, `3.14`) или по показателю степени (например, `314e-2`).
- Целочисленные литералы по умолчанию являются десятичными (например, `42` означает сорок два). Префикс `0b` указывает на двоичный (по основанию 2) целочисленный литерал (например, `0b10101010`). Префикс `0x` указывает на шестнадцатеричный (по основанию 16) целочисленный литерал (например, `0xBAD12CE3`). Префикс `0` указывает на восьмеричный (по основанию 8) целочисленный литерал (например, `0334`).

Чтобы сделать длинные литералы более удобочитаемыми для людей, мы можем использовать одинарную кавычку (`'`) в качестве разделителя цифр. Например, `π` - это примерно `3.14159'26535'89793'23846'26433'83279'50288` или, если вы предпочитаете шестнадцатеричную систему счисления `0x3.243F'6A88'85A3'08D3`.

1.4.1 Арифметика

Арифметические операторы могут быть использованы для соответствующих комбинаций фундаментальных типов:

```

x + y          // plus
+x             // unary plus
x - y          // minus
-x             // unary minus
x * y          // multiply

```

```
x / y      // divide
x % y      // remainder (modulus) for integers
```

То же самое могут сделать операторы сравнения:

```
x == y      // equal
x != y      // not equal
x < y       // less than
x > y       // greater than
x <= y      // less than or equal
x >= y      // greater than or equal
```

Кроме того, имеются логические операторы:

```
x & y       // bitwise and
x | y       // bitwise or
x ^ y       // bitwise exclusive or
~x          // bitwise complement
x && y      // logical and
x || y      // logical or
!x          // logical not (negation)
```

Побитовый логический оператор выдает результат того же типа что и операнды, для которых операция была выполнена над каждым битом. Логические операторы **&&** и **||** просто возвращают **true** или **false** в зависимости от значений их операндов.

В присваиваниях и арифметических операциях C++ выполняет все необходимые преобразования между базовыми типами, чтобы их можно было свободно смешивать:

```
void some_function()    // function that doesn't return a value
{
    double d = 2.2;      // initialize floating-point number
    int i = 7;           // initialize integer
    d = d + i;           // assign sum to d
    i = d * i;           // assign product to i;
                        // beware: truncating the double d*i to an int
}
```

Преобразования, используемые в выражениях, называются обычными арифметическими преобразованиями и направлены на то, чтобы гарантировать, что выражения вычисляются с максимальной точностью их операндов. Например, сложение **double** и **int** вычисляется с использованием арифметики с плавающей запятой двойной точности.

Обратите внимание, что **=** - это оператор присваивания, а **==** проверяет на равенство. В дополнение к обычным арифметическим и логическим операторам, C++ предлагает более специфические операции для изменения переменной:

```
x += y      // x = x+y
++x         // increment: x = x+1
x -= y      // x = x-y
--x         // decrement: x = x-1
x *= y      // scaling: x = x*y
x /= y      // scaling: x = x/y
x %= y      // x = x%y
```

Эти операторы лаконичны, удобны и очень часто используются.

Порядок вычисления слева направо для **x.y**, **x->y**, **x(y)**, **x[y]**, **x << y**, **x >> y**, **x && y** и **x || y**. Для присваиваний (например, **x += y**) порядок - справа налево. По историческим причинам, связанным с оптимизацией, порядок вычисления других выражений (например, **f(x) + g(y)**) и аргументов функции (например, **h(f(x), g(y))**), к сожалению, не определен.

1.4.2 Инициализация

Прежде чем объект можно будет использовать, ему должно быть присвоено значение. C++ предлагает различные варианты записи для выражения инициализации, таких как `=`, используемое выше, и универсальную форму, основанную на списках инициализаторов, заключённых в фигурные скобки:

```
double d1 = 2.3;           // initialize d1 to 2.3
double d2 {2.3};           // initialize d2 to 2.3
double d3 = {2.3};         // initialize d3 to 2.3 (the = is optional
                           // with { ... })
complex<double> z = 1;      // a complex number with double-precision
                           // floating-point scalars
complex<double> z2 {d1,d2};
complex<double> z3 = {d1,d2}; // the = is optional with { ... }

vector<int> v {1, 2, 3, 4, 5, 6}; // a vector of ints
```

Форма `=` является традиционной и восходит к C, но, если вы сомневаетесь, используйте общую форму списка инициализаторов `{}`. По крайней мере, это избавит вас от приведений типов, при которых теряется информация:

```
int i1 = 7.8;              // i1 becomes 7 (surprise?)
int i2 {7.8};              // error: floating-point to integer conversion
```

К сожалению, преобразования, которые теряют информацию, *сужающие преобразования*, такие как из `double` в `int` и из `int` в `char`, разрешены и неявно применяются при использовании `=` (но не при использовании `{}`). Проблемы, вызванные неявными сужающими преобразованиями, являются платой за совместимость с C (§19.3).

Константы (§1.6) нельзя оставлять неинициализированными, а переменную следует оставлять неинициализированной только в крайне редких случаях. Не вводите имя до тех пор, пока у вас не будет подходящего значения для него. Пользовательские типы (такие как `string`, `vector`, `Matrix`, `Motor_controller` или `Orc_warrior`) могут быть определены как неявно инициализируемые (§5.2.1).

При определении переменной вам не нужно явно указывать ее тип, если тип может быть выведен из инициализатора:

```
auto b = true;             // a bool
auto ch = x ;              // a char
auto i = 123;              // an int
auto d = 1.2;              // a double
auto z = sqrt(y);          // z has the type of whatever sqrt(y) returns
auto bb {true};            // bb is a bool
```

При использовании `auto` мы склонны использовать `=`, поскольку здесь не требуется потенциально сложное преобразование типов, но, если вы предпочитаете последовательно использовать инициализацию `{}`, вы можете сделать это.

Мы используем `auto` там, где у нас нет конкретной причины явно указывать тип. “Конкретные причины” включают:

- Определение находится в большой области видимости, где мы хотим сделать тип четко видимым для читателей нашего кода.
- Тип инициализатора неочевиден.
- Мы хотим четко указать диапазон или точность переменной (например, `double`, а не `float`)

Используя `auto`, мы избегаем избыточности и написания длинных имен типов. Это особенно важно в обобщённом программировании, где программисту может быть трудно определить точный тип объекта, а имена типов могут быть довольно длинными (§13.2).

1.5 Область видимости и время жизни

Объявление вводит объявляемое имя в область видимости:

- *Локальная область видимости*: имя, объявленное в функции (§1.3) или лямбда-выражении (§7.3.2), называется *локальным именем*. Его область действия простирается от точки объявления до конца блока, в котором находится его объявление. Блок ограничен парой `{}`. Имена аргументов функций считаются локальными именами.
- *Область видимости класса*: имя называется *именем члена* (или *именем члена класса*), если оно определено в классе (§2.2, §2.3, глава 5), вне какой-либо функции (§1.3), лямбда-выражения (§7.3.2) или перечисления `enum class` (§2.4). Его область видимости простирается от открывающей `{` его объявления до ближайшей `}`.
- *Область пространства имен*: Имя называется *именем члена пространства имен*, если оно определено в пространстве имен (§3.3) вне какой-либо функции, лямбды (§7.3.2), класса (§2.2, §2.3, глава 5) или `enum class` (§2.4). Его область действия простирается от точки объявления до конца его пространства имен.

Имя, не объявленное внутри какой-либо другой конструкции, называется *глобальным именем* и находится в *глобальном пространстве имен*.

Кроме того, у нас могут быть объекты без имен, такие как временные объекты и объекты, созданные с помощью `new` (§5.2.2). Например:

```
vector<int> vec;           // vec is global (a global vector of integers)

void fct(int arg)         // fct is global (names a global function)
                          // arg is local (names an integer argument)
{
    string motto {"Who dares wins"};    // motto is local
    auto p = new Record{"Hume"};        // p points to an unnamed Record (created by
new)
    // ...
}

struct Record {
    string name;    // name is a member of Record (a string member)
    // ...
};
```

Объект должен быть создан (инициализирован) до того, как он будет использован, и будет уничтожен при выходе из его области видимости. Для объекта пространства имен точкой уничтожения является завершение программы. Для члена точка разрушения определяется точкой разрушения объекта, членом которого он является. Объект, созданный с помощью `new`, “живет” до тех пор, пока не будет уничтожен с помощью `delete` (§5.2.2).

1.6 Константы

C++ поддерживает два понятия неизменяемости (объект с неизменяемым состоянием):

- **const**: примерно означает “Я обещаю не изменять это значение”. Это используется в первую очередь для указания интерфейсов, чтобы данные можно было передавать функциям с помощью указателей и ссылок, не опасаясь их изменения. Компилятор проверяет исполнение обещания, данного **const**. Значение **const** может быть вычислено во время выполнения.
- **constexpr**: примерно означает “вычисляется во время компиляции”. Это ключевое слово используется в первую очередь для указания констант, чтобы разрешить размещение данных в памяти, доступной только для чтения (где они вряд ли будут повреждены), и для повышения производительности. Значение **constexpr** должно быть вычислено компилятором.

Например:

```
constexpr int dmv = 17;           // dmv is a named constant
int var = 17;                     // var is not a constant
const double sqv = sqrt(var);     // sqv is a named constant, possibly computed at run
time

double sum(const vector<double>&); // sum will not modify its argument (§1.7)

vector<double> v {1.2, 3.4, 4.5}; // v is not a constant
const double s1 = sum(v);         // OK: sum(v) is evaluated at run time
constexpr double s2 = sum(v);     // error: sum(v) is not a constant expression
```

Чтобы функцию можно было использовать в константном выражении, то есть в выражении, которое будет вычисляться компилятором, она должна быть определена **constexpr** или **constexpr**. Например:

```
constexpr double square(double x) { return x * x; }

constexpr double max1 = 1.4*square(17); // OK: 1.4*square(17) is a constant expres-
sion
constexpr double max2 = 1.4*square(var); // error: var is not a constant,
// so square(var) is not a constant
const double max3 = 1.4*square(var);     // OK: may be evaluated at run time
```

Функция **constexpr** может использоваться для не константных аргументов, но, когда это делается, результат является не константным выражением. Мы разрешаем вызывать функцию **constexpr** с не константными аргументами в контекстах, которые не требуют константных выражений. Таким образом, нам не нужно определять по существу одну и ту же функцию дважды: один раз для константных выражений и один раз для переменных. Когда мы хотим, чтобы функция использовалась только для вычисления во время компиляции, мы объявляем ее **constexpr**, а не **constexpr**. Например:

```
constexpr double square2(double x) { return x*x; }
constexpr double max1 = 1.4*square2(17); // OK: 1.4*square(17) is a constant expres-
sion
const double max3 = 1.4*square2(var);    // error: var is not a constant
```

Функции, объявленные **constexpr** или **constexpr**, являются версией понятия *чистых функций* в C++. Они не могут иметь побочных эффектов и могут использовать только информацию, переданную им в качестве аргументов. В частности, они не могут изменять нелокальные переменные, но они могут иметь циклы и использовать свои собственные локальные переменные. Например:

```
constexpr double nth(double x, int n)    // assume 0<=n
{
    double res = 1;
    int i = 0;
    while (i < n) {    // while-loop: do while the condition is true (§1.7.1)
        res *= x;
        ++i;
    }
    return res;
}
```

В ряде мест константные выражения требуются языковыми правилами (например, для границ массива (§1.7), литералы оператора `case` (§1.8), аргументы значения шаблона (§7.2) и констант, объявленных с помощью `constexpr`). В других случаях вычисления времени компиляции важны для повышения производительности. Независимо от проблем с производительностью, понятие неизменяемости (объекта с неизменяемым состоянием) является важной концепцией проектирования.

1.7 Указатели, Массивы и Ссылки

Наиболее фундаментальный контейнер данных - это непрерывно расположенная последовательность элементов одного и того же типа, называемая *массивом*. Это в основном то, что предлагает аппаратное обеспечение. Массив элементов типа `char` может быть объявлен следующим образом:

```
char v[6];    // array of 6 characters
```

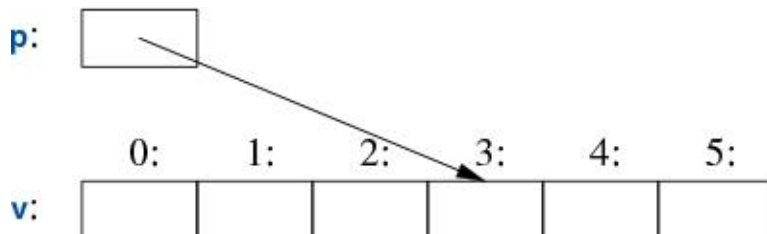
Аналогично, указатель может быть объявлен следующим образом:

```
char* p;    // pointer to character
```

В объявлениях `[]` означает “массив из”, а `*` - “указатель на”. Все массивы имеют `0` в качестве нижней границы, поэтому `v` содержит шесть элементов, от `v[0]` до `v[5]`. Размер массива должен быть константным выражением (§1.6). Переменная-указатель может содержать адрес объекта соответствующего типа:

```
char* p = &v[3];    // p points to v's fourth element
char x = *p;    // *p is the object that p points to
```

В выражении префикс унарная `*` означает “содержимое”, а префикс унарный `&` означает “адрес”. Мы можем представить это графически следующим образом:



Рассмотрим возможность вывода элементов массива:

```
void print()
{
    int v1[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    for (auto i=0; i!=10; ++i)    // print elements
        cout << v1[i] << '\n';
}
```

```

    //...
}

```

Это выражение **for** можно прочесть как “установите **i** равным нулю; пока **i** не равно **10**, выведите **i**-й элемент и увеличьте **i**”. С++ также предлагает более простой оператор **for**, называемый **for** для диапазона, для циклов, которые просто перебирают все элементы последовательности:

```

void print2()
{
    int v[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    for (auto x : v)                // for each x in v
        cout << x << '\n';

    for (auto x : {10, 21, 32, 43, 54, 65}) // for each integer in the list
        cout << x << '\n';
    //...
}

```

Первый оператор **for** для диапазона можно прочесть как “для каждого элемента **v**, от первого до последнего, поместите копию в **x** и выведите ее”. Обратите внимание, что нам не нужно указывать границы массива, когда мы инициализируем его списком. Оператор **for** для диапазона может использоваться для любой последовательности элементов (§13.1).

Если бы мы не хотели копировать значения из **v** в переменную **x**, а просто чтобы **x** ссылался на элемент, мы могли бы написать:

```

void increment()
{
    int v[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    for (auto& x : v)                // add 1 to each x in v
        ++x;
    //...
}

```

В объявлении унарный суффикс **&** означает “ссылка на”. Ссылка похожа на указатель, за исключением того, что вам не нужно использовать префикс ***** для доступа к значению, на которое указывает ссылка. Кроме того, ссылка после её инициализации не может быть изменена для указания на другой объект.

Ссылки особенно полезны для указания аргументов функции. Например:

```

void sort(vector<double>& v);        // sort v (v is a vector of doubles)

```

Используя ссылку, мы гарантируем, что для вызова **sort(my_vec)** мы не копируем **my_vec**. Следовательно, на самом деле сортируется **my_vec**, а не его копия.

Когда мы не хотим изменять аргумент, но хотим избежать затрат на копирование, мы используем **const** ссылку (§1.6); то есть ссылку на **const**. Например:

```

double sum(const vector<double>&)

```

Функции, принимающие константные ссылки, очень распространены.

При использовании в объявлениях операторы (такие как **&**, ***** и **[]**) называются *операторами деклараторов*:

```

T a[n]        // T[n]: a is an array of n Ts
T* p          // T*: p is a pointer to T
T& r          // T&: r is a reference to T

```

T $f(A)$ // $T(A)$: f is a function taking an argument of type A returning a result of type T

1.7.1 Нулевой указатель

Мы стараемся гарантировать, что указатель всегда указывает на объект, чтобы разыменовывание его было допустимым. Когда у нас нет объекта, на который можно указать, или если нам нужно представить понятие “объект недоступен” (например, для конца списка), мы присваиваем указателю значение `nullptr` (“нулевой указатель”). Существует только один `nullptr`, общий для всех типов указателей:

```
double* pd = nullptr;
Link<Record>* lst = nullptr; // pointer to a Link to a Record
int x = nullptr;           // error: nullptr is a pointer not an integer
```

Часто бывает разумно проверить, что аргумент указателя действительно указывает на что-то:

```
int count_x(const char* p, char x)
// count the number of occurrences of x in p[]
// p is assumed to point to a zero-terminated array of char (or to nothing)
{
    if (p==nullptr)
        return 0;
    int count = 0;
    for (; *p!=0; ++p)
        if (*p==x)
            ++count;
    return count;
}
```

Мы можем сместить указатель, чтобы он указывал на следующий элемент массива, используя `++`, а также опустить инициализатор в операторе `for`, если он нам не нужен.

Определение `count_x()` предполагает, что `char*` является строкой в стиле C, то есть указатель указывает на массив `char`, заканчивающийся нулем. Символы в строковом литерале неизменяемы, поэтому для возможности вызова `count_x("Hello!")` я объявил в `count_x()` константный аргумент `const char*`.

В старом коде обычно используется `0` или `NULL` вместо `nullptr`. Однако использование `nullptr` устраняет потенциальную путаницу между целыми числами (такими как `0` или `NULL`) и указателями (такими как `nullptr`).

В примере `count_x()` мы не используем часть оператора `for` предназначенную для инициализатора, поэтому мы можем использовать более простой оператор `while`:

```
int count_x(const char* p, char x)
// count the number of occurrences of x in p[]
// p is assumed to point to a zero-terminated array of char (or to nothing)
{
    if (p==nullptr)
        return 0;
    int count = 0;
    while (*p) {
        if (*p==x)
            ++count;
        ++p;
    }
    return count;
}
```

Оператор **while** выполняется до тех пор, пока его условие не станет ложным (**false**).

Проверка числового значения (например, **while (*p)** в **count_x()**) эквивалентна сравнению значения с **0** (например, **while (*p!=0)**). Проверка значения указателя (например, **if (p)**) эквивалентна сравнению значения с **nullptr** (например, **if (p!=nullptr)**).

Нет никакой “нулевой ссылки”. Ссылка должна ссылаться на допустимый объект (и реализации предполагают, что это так). Существуют неясные и хитроумные способы нарушить это правило; но не делайте этого.

1.8 Условные операторы

C++ предоставляет обычный набор операторов для ветвлений и циклов, таких как **if**, **switch**, **while** и **for**. Например, вот простая функция, которая запрашивает пользовательский ввод и возвращает логическое значение, соответствующее вводу:

```
bool accept()
{
    cout << "Do you want to proceed (y or n)?\n";    // write question
    char answer = 0;                                // initialize to a value
                                                    // that will not appear on input
    cin >> answer;                                  //read answer

    if (answer == 'y' )
        return true;
    return false;
}
```

Аналогично оператору вывода **<<** (“put to”), для ввода используется оператор **>>** (“get from”); **cin** - стандартный входной поток ([глава 11](#)). Тип правого операнда **>>** определяет, какой ввод принимается, а его правый операнд является целью операции ввода. Символ **\n** в конце выходной строки обозначает собой переход на новую строку (§1.2.1).

Обратите внимание, что определение **answer** появляется там, где это необходимо (а не до этого). Объявление может появиться в любом месте, где может появиться оператор.

Пример можно было бы улучшить, приняв во внимание **n** (для ответа “нет”):

```
bool accept2()
{
    cout << "Do you want to proceed (y or n)?\n";    // write question
    char answer = 0;                                // initialize to a value
                                                    // that will not appear on input
    cin >> answer;                                  // read answer

    switch (answer) {
        case 'y':
            return true;
        case 'n':
            return false;
        default:
            cout << "I'll take that for a no.\n";
            return false;
    }
}
```

Оператор **switch** сравниваем значение на соответствие набору констант. Эти константы, называемые **case**-метками, должны быть разными, и, если проверяемое значе-

ние не соответствует ни одной из них, выбирается метка **default**. Если значение не соответствует ни одной **case**-метке и **default**-метка не указана, никакие действия не предпринимаются.

Нам не нужно выходить из **case** при помощи **return** как из функции, которая содержит его оператор **switch**. Часто мы просто хотим продолжить выполнение с инструкции, следующей за оператором **switch**. Мы можем сделать это, используя оператор **break**. В качестве примера рассмотрим чрезмерно умный, но примитивный синтаксически анализатор для простейших команд видеоигры:

```
void action()
{
    while (true) {
        cout << "enter action:\n";           // request action
        string act;
        cin >> act;                           // read characters into a string
        Point delta {0,0};                     // Point holds an {x,y} pair

        for (char ch : act) {
            switch (ch) {
                case 'u': // up
                case 'n': // north
                    ++delta.y;
                    break;
                case 'r': // right
                case 'e': // east
                    ++delta.x;
                    break;
                // ... more actions ...
                default:
                    cout << "I freeze!\n";
            }
            move(current + delta * scale);
            update_display();
        }
    }
}
```

Подобно оператору **for** (§1.7), оператор **if** может вводить переменную и проверять ее. Например:

```
void do_something(vector<int>& v)
{
    if (auto n = v.size(); n!=0) {
        // ... we get here if n!=0 ...
    }
    // ...
}
```

Здесь целое число **n** определено для использования в операторе **if**, инициализируется с помощью **v.size()** и немедленно проверяется условием **n!=0** после точки с запятой. Имя, объявленное в условии, находится в области видимости в обеих ветвях оператора **if**.

Как и в случае с оператором **for**, целью объявления имени в условии оператора **if** является ограничение области видимости переменной для улучшения удобочитаемости и минимизации ошибок.

Наиболее распространенным случаем является проверка переменной на равенство **0** (или **nullptr**). Чтобы сделать это, просто опустите явное упоминание об этом условии. Например:

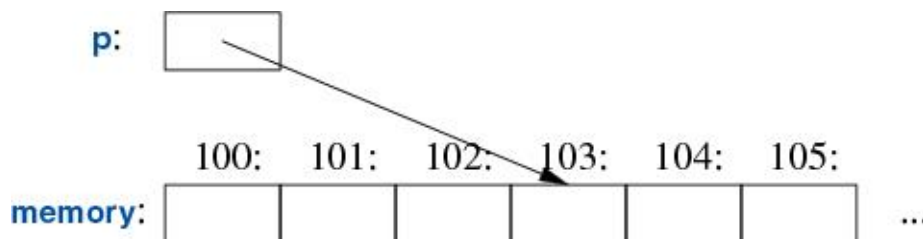

```
void do_something(vector<int>& v)
{
    if (auto n = v.size()) {
        // ... we get here if n!=0 ...
    }
    // ...
}
```

Предпочтительно использование этой краткой и простой формы, когда можете.

1.9 Сопоставление с аппаратным обеспечением

C++ предлагает прямое сопоставление с аппаратным обеспечением. Когда вы используете одну из основных операций, реализация - это то, что предлагает аппаратное обеспечение, обычно это одна машинная операция. Например, при сложении двух `int`, `x+y` выполняет машинную инструкцию сложения целых чисел.

Реализация C++ рассматривает память машины как последовательность ячеек памяти, в которые она может размещать (типизированные) объекты и обращаться к ним с помощью указателей:



Указатель представлен в памяти в виде машинного адреса, поэтому числовое значение `p` на этом рисунке будет равно `103`. Если это очень похоже на массив (§1.7), то это потому, что массив - это базовая абстракция C++, представляющая собой “непрерывную последовательность объектов в памяти”.

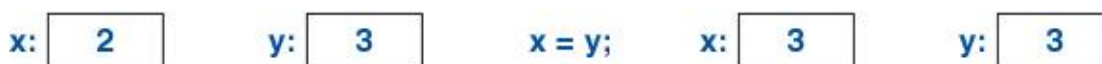
Простое сопоставление фундаментальных языковых конструкций с аппаратным обеспечением имеет решающее значение для высокой производительности, которой C и C++ славятся десятилетиями. Базовая машинная модель C и C++ основана на компьютерном оборудовании, а не на какой-либо математической абстракции.

1.9.1 Присваивание

Присваивание встроенного типа - это простая операция машинного копирования. Рассмотрим:

```
int x = 2;
int y = 3;
x = y;           // x becomes 3; so we get x==y
```

Мы можем наглядно представить это графически следующим образом:



Эти два объекта независимы. Мы можем изменить значение `y`, не влияя на значение `x`. Например, `x=99` не изменит значение `y`. В отличие от Java, C# и других языков, но подобно Си, это верно для всех типов, а не только для `int`.

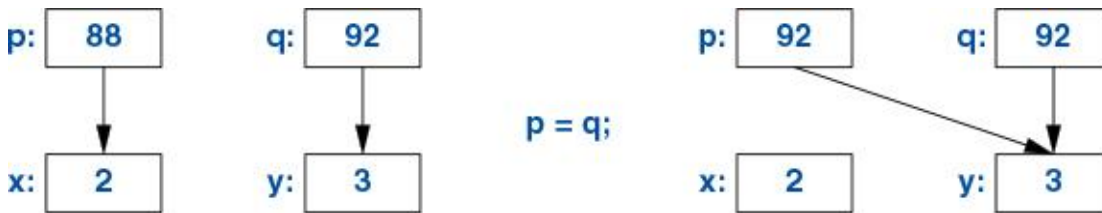
Если мы хотим, чтобы разные объекты ссылались на одно и то же (общее) значение, мы должны написать следующее:

```

int x = 2;
int y = 3;
int* p = &x;
int* q = &y;           // p!=q and *p!=*q
p = q;                 // p becomes &y; now p==q, so (obviously)*p==*q

```

Можно представить это графически следующим образом:



Я произвольно выбрал 88 и 92 в качестве адресов `int`. Опять же, мы можем видеть, что объект, которому присвоено значение, получает значение от назначенного объекта, в результате чего получаются два независимых объекта (здесь указатели) с одинаковым значением. То есть `p=q` дает `p==q`. После `p=q` оба указателя указывают на `y`.

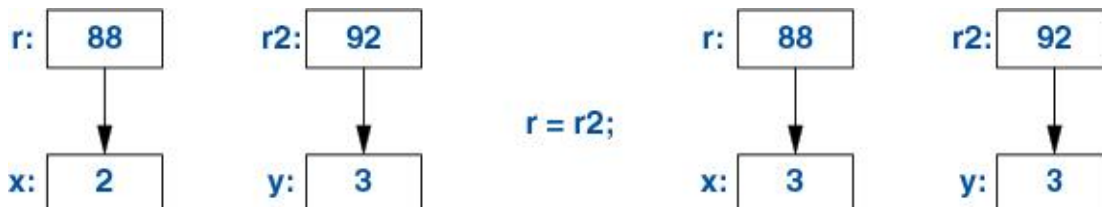
Ссылка и указатель оба ссылаются/указывают на объект, и оба представлены в памяти в виде машинного адреса. Однако языковые правила их использования различаются. Присвоение ссылки не изменяет адрес объекта, на который указывает ссылка, а присваивает объекту, на который указывает ссылка:

```

int x = 2;
int y = 3;
int& r = x;           // r refers to x
int& r2 = y;          // r2 refers to y
r = r2;               // read through r2, write through r: x becomes 3

```

Мы можем представить это графически следующим образом:



Чтобы получить доступ к значению, на которое указывает указатель, вы используете `*`; для ссылки это выполняется неявно.

После `x=y` у выполняется `x==y` для каждого встроенного типа и хорошо спроектированного пользовательского типа ([глава 2](#)), для которых реализованы операторы `=` (присвоение) и `==` (сравнение на равенство).

1.9.2 Инициализация

Инициализация отличается от присваивания. В общем случае, чтобы присваивание работало корректно, объект, которому присвоено значение, должен иметь значение. С другой стороны, задача инициализации состоит в том, чтобы превратить неинициализированный фрагмент памяти в корректный объект. Почти для всех типов результат чтения из неинициализированной переменной или записи в нее не определен.

Рассмотрим примеры:

```

int x = 7;
int& r {x};           // bind r to x (r refers to x)
r = 7;                // assign to whatever r refers to

```

```
int& r2;           // error: uninitialized reference
r2 = 99;           // assign to whatever r2 refers to
```

К счастью, у нас не может быть неинициализированной ссылки; если бы мы могли, то при `r2=99` присвоили бы `99` некоторой неопределенной ячейке памяти; результат в конечном итоге привел бы к неверным результатам или сбою.

Вы можете использовать `=` для инициализации ссылки, но, пожалуйста, пусть это вас не смущает. Например:

```
int& r = x;        // bind r to x (r refers to x)
```

Это все еще инициализация и привязывает `r` к `x`, а не какая-либо форма копирования значения.

Различие между инициализацией и присваиванием также имеет решающее значение для многих пользовательских типов, таких как `string` и `vector`, где целевой объект присваивания владеет ресурсом, который в конечном итоге должен быть освобожден (§6.3).

Основная семантика передачи аргументов и возврата значения из функции - это семантика инициализации (§3.4). Например, так мы получаем возможность передачи аргументов по ссылке (§3.4.1).

1.10 Советы

Приведенные здесь рекомендации являются подмножеством руководства C++ Core Guidelines [Stroustrup,2015]. Ссылки на руководство выглядят следующим образом [CG: ES.23], что означает 23-е правило в разделе "Выражения и утверждения". Как правило, основное руководство содержит дополнительные обоснования и примеры.

- [1] Не паникуйте! Все станет ясно со временем; §1.1; [CG: In.0].
- [2] Не используйте исключительно встроенные функции. Многие фундаментальные (встроенные) функции обычно лучше всего использовать косвенно через библиотеки, такие как стандартная библиотека ISO C++ (Chapters 9–18); [CG: P.13].
- [3] Используйте `#include` или (предпочтительнее) `import` библиотек, необходимых для упрощения программирования; §1.2.1.
- [4] Вам не обязательно знать все детали C++, чтобы писать хорошие программы.
- [5] Сосредоточьтесь на методах программирования, а не на возможностях языка.
- [6] Стандарт ISO C++ является последним словом в вопросах определения языка; §19.1.3; [CG: P.2].
- [7] "Упаковывайте" значимые операции в виде тщательно названных функций; §1.3; [CG: F.1].
- [8] Функция должна выполнять единственную смысловую операцию; §1.3 [CG: F.2].
- [9] Делайте функции короче; §1.3; [CG: F.3].
- [10] Используйте перегрузку, когда функции выполняют концептуально одну и ту же задачу для разных типов; §1.3.
- [11] Если функция может быть вычислена во время компиляции, объявите ее `constexpr`; §1.6; [CG: F.4].
- [12] Если функция должна быть вычислена во время компиляции, объявите ее `constexpr`; §1.6.
- [13] Если функция может не иметь побочных эффектов, объявите ее `constexpr` или `constexpr`; §1.6; [CG: F.4].

- [14] Поймите, как языковые примитивы соотносятся с аппаратным обеспечением; §1.4, §1.7, §1.9, §2.3, §5.2.2, §5.4.
- [15] Используйте разделители цифр, чтобы сделать большие литералы удобочитаемыми; §1.4; [CG: NL.11].
- [16] Избегайте сложных выражений; [CG: ES.40].
- [17] Избегайте сужающих преобразований типов; §1.4.2; [CG: ES.46].
- [18] Минимизируйте область видимости переменной; §1.5, §1.8.
- [19] Держите области видимости маленькими; §1.5; [CG: ES.5].
- [20] Избегайте “магических констант”; используйте символические константы; §1.6; [CG: ES.45].
- [21] Предпочтительнее неизменяемые данные; §1.6; [CG: P.10].
- [22] Объявляйте только одно имя при каждом объявлении; [CG: ES.10].
- [23] Выбирайте обычные и локальные имена короче; необычные и нелокальные имена длиннее; [CG: ES.7].
- [24] Избегайте похожих имён; [CG: ES.8].
- [25] Избегайте **ALL_CAPS** имён; [CG: ES.9].
- [26] Предпочитайте синтаксис **{}**-инициализатора для объявлений с именованным типом; §1.4; [CG: ES.23].
- [27] Используйте **auto**, чтобы избежать повторения имен типов; §1.4.2; [CG: ES.11].
- [28] Избегайте неинициализированных переменных; §1.4; [CG: ES.20].
- [29] Не объявляйте переменную до тех пор, пока у вас не будет значения для ее инициализации; §1.7, §1.8; [CG: ES.21].
- [30] При объявлении переменной в условии оператора **if** отдавайте предпочтение версии с неявной проверкой на **0** or **nullptr**; §1.8.
- [31] Предпочитайте циклы **for** для диапазонов циклам **for** с явной переменной цикла; §1.7.
- [32] Используйте **unsigned** только для битовых манипуляций; §1.4; [CG: ES.101] [CG: ES.106].
- [33] Старайтесь использовать указатели простыми и прямолинейными; §1.7; [CG: ES.42].
- [34] Использование **nullptr** предпочтительнее чем **0** или **NULL**; §1.7; [CG: ES.47].
- [35] Не пишите в комментариях то, что может быть четко указано в коде; [CG: NL.1].
- [36] Укажите в комментариях свои намерения; [CG: NL.2].
- [37] Поддерживайте единый стиль отступов; [CG: NL.4].

Пользовательские типы

Не паникуй!
– Дуглас Адамс

- [Введение](#)
- [Структуры](#)
- [Классы](#)
- [Перечисления](#)
- [Объединения](#)
- [Советы](#)

2.1 Введение

Мы называем типы, которые могут быть построены на основе фундаментальных типов (§1.4), модификатора `const` (§1.6) и операторов объявления (§1.7), *встроенными типами*. Набор встроенных типов и операций в C++ богат, но намеренно сделан низкоуровневым. Они непосредственно и эффективно отражают возможности обычного компьютерного оборудования. Однако они не предоставляют программисту высокоуровневых возможностей для удобного написания сложных приложений. Вместо этого C++ дополняет встроенные типы и операции сложным набором *механизмов абстракции*, с помощью которых программисты могут создавать необходимые высокоуровневые средства.

Механизмы абстракции C++ в первую очередь предназначены для того, чтобы позволить программистам разрабатывать и реализовывать свои собственные типы с соответствующими представлениями и операциями, и просто и элегантно использовать такие типы. Типы, созданные из других типов с использованием механизмов абстракции C++, называются *пользовательскими типами*. Они называются *классами* и *перечислениями*. Пользовательские типы могут быть созданы как из встроенных типов, так и из других пользовательских типов. Большая часть этой книги посвящена проектированию, реализации и использованию пользовательских типов. Пользовательские типы часто предпочтительнее встроенных типов, потому что они проще в использовании, менее подвержены ошибкам и, как правило, столь же эффективны в том, что они делают, как прямое использование встроенных типов, или даже более эффективны.

В остальной части этой главы представлены самые простые и фундаментальные средства определения и использования типов. [Главы 4-8](#) представляют собой более полное описание механизмов абстракции и поддерживаемых ими стилей программирования. Пользовательские типы составляют основу стандартной библиотеки, поэтому

в [главах 9-17](#), посвященных стандартной библиотеке, приведены примеры того, что может быть создано с использованием языковых средств и методов программирования, представленных в [главах 1-8](#).

2.2 Структуры

Первым шагом в создании нового типа часто является организация необходимых ему элементов в структуру данных `struct`:

```
struct Vector {
    double* elem;    // pointer to elements
    int sz;          // number of elements
};
```

Эта первая версия `Vector` состоит из `int` и `double*`.

Переменная типа `Vector` может быть определена следующим образом:

```
Vector v;
```

Однако само по себе это не имеет большого значения, потому что указатель `elem` объекта `v` ни на что не указывает. Чтобы это можно было использовать, мы должны присвоить `v` указатель на некоторые элементы. Например:

```
void vector_init(Vector& v, int s)    // initialize a Vector
{
    v.elem = new double[s];           // allocate an array of s doubles
    v.sz = s;
}
```

То есть поле `elem` объекта `v` получает указатель, созданный оператором `new`, а поле `sz` объекта `v` получает количество элементов. Символ `&` в `Vector&` указывает, что мы передаем объект `v` по неконстантной ссылке (§1.7); таким образом, функция `vector_init()` может изменить переданный ей вектор.

Оператор `new` выделяет память из области, называемой *динамической памятью* (также известной также как *куча* или *свободное хранилище*). Объекты, размещенные в динамической памяти, не зависят от области видимости, из которой они созданы, и “живут” до тех пор, пока не будут уничтожены с помощью оператора `delete` (§5.2.2).

Простое использование `Vector` выглядит следующим образом:

```
double read_and_sum(int s)
    // read s integers from cin and return their sum; s is assumed to be positive
{
    Vector v;
    vector_init(v,s);                // allocate s elements for v

    for (int i=0; i!=s; ++i)
        cin>>v.elem[i];             // read into elements

    double sum = 0;
    for (int i=0; i!=s; ++i)
        sum+=v.elem[i];              // compute the sum of the elements
    return sum;
}
```

Предстоит пройти долгий путь, прежде чем наш `Vector` станет таким же элегантным и гибким, как `vector` стандартной библиотеки. В частности, пользователь `Vector` должен

знать каждую деталь представления **Vector**. Остальная часть этой главы и две следующие постепенно улучшают **Vector** в качестве примера языковых возможностей и техник. В [главе 12](#) представлен **vector** стандартной библиотеки, который содержит множество приятных улучшений.

Я использую **vector** и другие компоненты стандартной библиотеки в качестве примеров чтобы:

- проиллюстрировать особенности языка и методы проектирования, а также
- помочь вам изучить и использовать компоненты стандартной библиотеки.

Не изобретайте заново компоненты стандартной библиотеки, такие как **vector** и **string**; используйте их. Типы стандартной библиотеки имеют имена в нижнем регистре, поэтому, чтобы различать названия типов, используемых для иллюстрации методов проектирования и реализации в качестве примеров (например, **Vector** и **String**), я пишу их с заглавной буквы.

Для доступа к элементам структуры **struct** через имя (и через ссылку) мы используем **.** (точку), а для доступа через указатель - оператор **->**. Например:

```
void f(Vector v, Vector& rv, Vector* pv)
{
    int i1 = v.sz;           // access through name
    int i2 = rv.sz;          // access through reference
    int i3 = pv->sz;         // access through pointer
}
```

2.3 Классы

Определение данных отдельно от операций над ними имеет преимущества, такие как возможность использовать данные произвольными способами. Однако для того, чтобы пользовательский тип обладал всеми свойствами, ожидаемыми от “реального типа”, необходима более тесная связь между представлением данных и операциями над ними. В частности, мы часто хотим сохранить представление данных недоступным для пользователей, чтобы упростить использование, гарантировать согласованное использование данных и позволить нам позже улучшить представление данных. Чтобы сделать это, мы должны различать интерфейс типа (который общедоступен) и его реализацию (которая имеет доступ к данным недоступным другими способами). Языковой механизм для этого называется классом. Класс состоит из набора членов, в котором могут быть элементы данных, функции или типы.

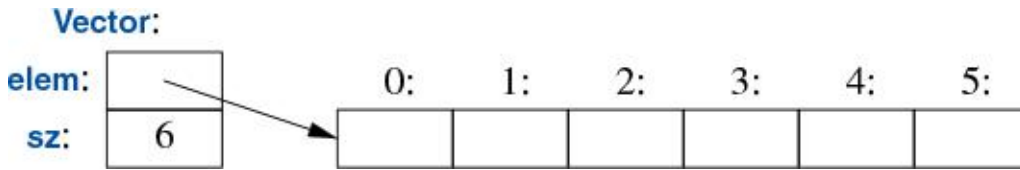
Интерфейс класса определяется его **public** публичными членами, а его **private** приватные члены доступны только через этот интерфейс. Публичная **public** и приватная **private** части объявления класса могут отображаться в любом порядке, но обычно мы размещаем **public** объявления первыми, а **private** - позже, за исключением случаев, когда мы хотим подчеркнуть представление. Например:

```
class Vector {
public:
    Vector(int s) :elem{new double[s]}, sz{s} { } // construct a Vector
    double& operator[](int i) { return elem[i]; } // element access: subscripting
    int size() { return sz; }
private:
    double* elem; // pointer to the elements
    int sz;       // the number of elements
};
```

Учитывая это, мы можем определить переменную нашего нового типа **Vector**:

```
Vector v(6);      // a Vector with 6 elements
```

Мы можем проиллюстрировать объект **Vector** графически:



По сути, объект **Vector** представляет собой “дескриптор”, содержащий указатель на элементы (**elem**) и количество элементов (**sz**). Количество элементов (6 в примере) может варьироваться от одного объекта **Vector** к другому объекту **Vector**, и объект **Vector** может иметь разное количество элементов в разное время (§5.2.3). Однако сам объект **Vector** всегда имеет одинаковый размер. Это основной метод обработки различных объемов информации в C++: дескриптор фиксированного размера, ссылающийся на переменный объем данных “в другом месте” (например, в динамической памяти, выделенной **new**; §5.2.2). Как проектировать и использовать такие объекты - основная тема главы 5.

Здесь представление **Vector** (элементы **elem** и **sz**) доступно только через интерфейс, предоставляемый **public** элементами: **Vector()**, **operator[]()**, и **size()**. Пример **read_and_sum()** из §2.2 упрощается до:

```
double read_and_sum(int s)
{
    Vector v(s);                // make a vector of s elements
    for (int i=0; i!=v.size(); ++i)
        cin>>v[i];             // read into elements

    double sum = 0;
    for (int i=0; i!=v.size(); ++i)
        sum+=v[i];              // take the sum of the elements
    return sum;
}
```

Функция-член с тем же именем, что и у её класса, называется *конструктором*, то есть функцией, используемой для *создания* объектов класса. Таким образом, конструктор **Vector()** заменяет **vector_init()** из §2.2. В отличие от обычной функции, конструктор гарантированно используется для инициализации объектов своего класса. Таким образом, определение конструктора устраняет проблему неинициализированных переменных для класса.

Vector(int) определяет, как создаются объекты типа **Vector**. В частности, в нем указано, что для этого ему нужно целое число. Это целое число используется в качестве количества элементов. Конструктор инициализирует члены объекта **Vector**, используя список инициализаторов элементов:

```
:elem{new double[s]}, sz{s}
```

То есть сначала инициализируется **elem** указателем на **s** элементов типа **double**, полученных из кучи. Затем инициализируется **sz** значением **s**.

Доступ к элементам обеспечивается функцией индекса, называемой **operator[]**. Он возвращает ссылку на соответствующий элемент (**double&** позволяющий как чтение, так и запись).

Функция **size()** возвращает количество хранящихся элементов.

Очевидно, что обработка ошибок здесь полностью отсутствует, но мы вернемся к этому в [главе 4](#). Аналогично, мы не предоставили механизм для “возврата” массива `double`, полученного с помощью `new`; в §5.2.2 показано, как определить деструктор, чтобы элегантно сделать это.

Между `struct` и `class` нет принципиальной разницы; `struct` - это просто `class`, члены которого по умолчанию являются `public`. Например, вы можете определить конструкторы и другие функции-члены для `struct`.

2.4 Перечисления

В дополнение к классам, C++ поддерживает простую форму пользовательского типа, для которой мы можем перечислять значения:

```
enum class Color { red, blue, green };
enum class Traffic_light { green, yellow, red };

Color col = Color::red;
Traffic_light light = Traffic_light::red;
```

Обратите внимание, что перечислители (например, `red`) находятся в области видимости их `enum class`, так что их можно многократно использовать в разных `enum class` без путаницы. Например, `Color::red` - это `red` класса `Color`, который отличается от `Traffic_light::red`.

Перечисления используются для представления небольших наборов целочисленных значений. Перечисления нужны чтобы сделать код более читаемым и менее подверженным ошибкам, чем при использовании литералов, а не символических (и мнемонических) имён перечислителей.

`class` после `enum` указывает, что перечисление строго типизировано и что его перечислители ограничены пространством имён класса. Будучи отдельными типами, `enum class` помогают предотвратить случайное неправильное использование констант. В частности, мы не можем смешивать значения `Traffic_light` и `Color`:

```
Color x1 = red;           // error: which red?
Color y2 = Traffic_light::red; // error: that red is not a Color
Color z3 = Color::red;     // OK
auto x4 = Color::red;      // OK: Color::red is a Color
```

Аналогично, мы не можем неявно смешивать `Color` и целочисленные значения:

```
int i = Color::red;       // error: Color::red is not an int
Color c = 2;              // initialization error: 2 is not a Color
```

Перехват попыток преобразования в `enum` является хорошей защитой от ошибок, но часто мы хотим инициализировать `enum` значением из его базового типа (по умолчанию это `int`), так что это разрешено, как и явное преобразование из базового типа:

```
Color x = Color{5}; // OK, but verbose
Color y {6};        // also OK
```

Аналогично, мы можем явно преобразовать значение `enum` в его базовый тип:

```
int x = int(Color::red);
```

По умолчанию в `enum class` определены присваивание, инициализация и сравнения (например, `==` и `<`; §1.4), и только они. Однако перечисление - это пользовательский тип, поэтому мы можем определить для него операторы (§6.4):

```
Traffic_light& operator++(Traffic_light& t) // prefix increment: ++
{
```

```

switch (t) {
case Traffic_light::green:    return t=Traffic_light::yellow;
case Traffic_light::yellow:  return t=Traffic_light::red;
case Traffic_light::red:     return t=Traffic_light::green;
}
}

auto signal = Traffic_light::red;
Traffic_light next = ++signal;    // next becomes Traffic_light::green

```

Если повторение имени перечисления, **Traffic_light**, становится слишком утомительным, мы можем сократить его в области видимости:

```

Traffic_light& operator++(Traffic_light& t)    // prefix increment: ++
{
    using enum Traffic_light;                // here, we are using Traffic_light

    switch (t) {
        case green:    return t=yellow;
        case yellow:   return t=red;
        case red:      return t=green;
    }
}

```

Если вы никогда не захотите явно указывать имена перечислителей и хотите, чтобы значения перечислителя были **int** (без необходимости явного преобразования), вы можете удалить **class** из **enum class**, чтобы получить “простое” перечисление **enum**. Перечислители из “простого” **enum** вводятся в ту же область видимости, что и имя их **enum**, и не явно преобразуются в их целочисленные значения. Например:

```

enum Color { red, green, blue };
int col = green;

```

Здесь **col** получает значение **1**. По умолчанию целочисленные значения счетчиков начинаются с **0** и увеличиваются на единицу для каждого следующего перечислителя. “Простые” перечисления **enum** были в C++ (и C) с самых первых дней, поэтому, несмотря на то, что они ведут себя менее корректно, они распространены в текущем коде.

2.5 Объединения

Объединение **union** - это **struct**, в которой все члены располагаются по одному и тому же адресу, так что **union** занимает ровно столько места, сколько занимает его самый большой элемент. Естественно, **union** может содержать значение только для одного члена одновременно. Например, рассмотрим запись таблицы символов, которая содержит имя и значение. Значением может быть либо **Node***, либо **int**:

```

enum class Type { ptr, num };    // a Type can hold values ptr and num (§2.4)

struct Entry {
    string name;    // string is a standard-library type
    Type t;
    Node* p;        // use p if t==Type::ptr
    int i;          // use i if t==Type::num
};

void f(Entry* pe)
{
    if (pe->t == Type::num)
        cout << pe->i;
}

```

```

    // ...
}

```

Члены `p` и `i` никогда не используются одновременно, поэтому память тратится впустую. Это можно легко исправить, указав, что оба должны быть членами `union`, например так:

```

union Value {
    Node* p;
    int i;
};

```

Теперь `Value::p` и `Value::i` помещаются по одному и тому же адресу памяти каждого объекта `Value`.

Такого рода оптимизация пространства может быть важна для приложений, которые занимают большие объемы памяти, поэтому компактное представление имеет решающее значение.

Язык не отслеживает, какое значение содержит `union`, поэтому программист должен это сделать:

```

struct Entry {
    string name;
    Type t;
    Value v;    // use v.p if t==Type::ptr; use v.i if t==Type::num
};

void f(Entry* pe)
{
    if (pe->t == Type::num)
        cout << pe->v.i;
    // ...
}

```

Поддержание соответствия между *полем типа*, иногда называемым *дискриминантом* или *тегом* (здесь `t`), и типом, содержащимся в `union`, подвержено ошибкам. Чтобы избежать ошибок, мы можем обеспечить соблюдение этого соответствия, инкапсулировав объединение и поле типа в класс и предложив доступ только через функции-члены, которые правильно используют объединение. На прикладном уровне абстракции, опирающиеся на такие объединения с тегами, являются распространенными и полезными. Использование “голых” `union` лучше свести к минимуму.

В стандартной библиотеке тип `variant`, может быть использован для устранения большинства прямых применений объединений. `variant` хранит одно значение из набора альтернативных типов (§15.4.1). Например, `variant<Node*, int>` может содержать либо `Node*`, либо `int`. Используя `variant`, пример `Entry` может быть записан как:

```

struct Entry {
    string name;
    variant<Node*, int> v;
};

void f(Entry* pe)
{
    if (holds_alternative<int>(pe->v))    // does *pe hold an int? (see §15.4.1)
        cout << get<int>(pe->v);        // get the int
    // ...
}

```

Для многих применений `variant` проще и безопаснее в использовании, чем `union`.

2.6 Советы

- [1] Отдавайте предпочтение четко определенным пользовательским типам перед встроенными типами, если встроенные типы слишком низкоуровневые; §2.1.
- [2] Организуйте связанные данные в структуры (`struct` или `class`); §2.2; [CG: C.1].
- [3] Понимайте различие между интерфейсом и реализацией, использующей `class`; §2.3; [CG: C.3].
- [4] `struct` - это просто `class`, члены которого по умолчанию являются `public`; §2.3.
- [5] Определите конструкторы чтобы гарантировать и упростить инициализацию классов `class`; §2.3; [CG: C.2].
- [6] Используйте перечисления для представления наборов именованных констант; §2.4; [CG: Enum.2].
- [7] Предпочитайте `class enum` “простым” `enum`, чтобы свести к минимуму неожиданности; §2.4; [CG: Enum.3].
- [8] Определите операции над перечислениями для безопасного и простого использования; §2.4; [CG: Enum.4].
- [9] Избегайте “голых” `union`; оберните их в класс вместе с полем типа §2.5; [CG: C.181].
- [10] Предпочитайте `std::variant` “голым `union`”; §2.5.

Модульность

*Занимайся своим делом.
– американская поговорка*

- [Введение](#)
- [Раздельная компиляция](#)

[Заголовочные файлы](#); [Модули](#)

- [Пространства имён](#)
- [Аргументы функции и возвращаемые значения](#)

[Передача аргументов](#); [Возвращение значений](#); [Выведение типа возвращаемого значения](#); [Суффиксная запись типа возвращаемого значения](#); [Структурное связывание](#)

- [Советы](#)

3.1 Введение

Программа на C++ состоит из множества отдельно разработанных частей, таких как функции (§1.2.1), пользовательские типы (глава 2), иерархии классов (§5.5) и шаблоны (глава 7). Ключом к управлению таким множеством частей является четкое определение взаимодействий между этими частями. Первым и наиболее важным шагом является разделение интерфейса к части и ее реализации. На уровне языка C++ интерфейсы представлены с помощью объявлений. *Объявление* определяет все, что необходимо для использования функции или типа. Например:

```
double sqrt(double);    // the square root function takes a double and returns a double
class Vector {           // what is needed to use a Vector
public:
    Vector(int s);
    double& operator[](int i);
    int size();
private:
    double* elem;        // elem points to an array of sz doubles
    int sz;
};
```

Ключевым моментом здесь является то, что тела функций, *определения* функций, могут находиться “в другом месте”. В этом примере мы могли бы пожелать, чтобы представление `Vector` также было “в другом месте”, но мы разберемся с этим позже (говоря об абстрактных типах; §5.3). Определение `sqrt()` будет выглядеть следующим образом:


```
double sqrt(double d)           // definition of sqrt()
{
    // ... algorithm as found in math textbook ...
}
```

Для **Vector** нам нужно определить все три функции-члена:

```
Vector::Vector(int s)           // definition of the constructor
    :elem{new double[s]}, sz{s} // initialize members
{
}

double& Vector::operator[](int i) // definition of subscripting
{
    return elem[i];
}

int Vector::size()              // definition of size()
{
    return sz;
}
```

Мы должны определить функции **Vector**, но не **sqrt()**, потому что это часть стандартной библиотеки. Однако реальной разницы нет: библиотека - это просто “какой-то другой код, который мы используем при необходимости”, написанный с использованием тех же языковых средств, которые используем мы.

Для объекта, такого как функция, может быть много объявлений, но только одно определение.

3.2 Раздельная компиляция

C++ поддерживает концепцию раздельной компиляции, когда пользовательский код видит только объявления используемых типов и функций. Это можно сделать двумя способами:

- *Заголовочные файлы* (§3.2.1): Поместите объявления в отдельные файлы, называемые *заголовочными файлами*, и буквально **#include** (включите) заголовочный файл там, где необходимы его объявления.
- *Модули* (§3.2.2): Определите файлы модулей **module**, скомпилируйте их отдельно и импортируйте **import** их при необходимости. Код, импортирующий **import** модуль **module**, видит только явно экспортированные **export** объявления.

Любой из них может быть использован для организации программы в набор полунезависимых фрагментов кода. Такое разделение может быть использовано для минимизации времени компиляции и обеспечения разделения логически различных частей программы (таким образом, сводя к минимуму вероятность ошибок). Библиотека часто представляет собой набор отдельно скомпилированных фрагментов кода (например, функций).

Метод организации кода с использованием заголовочных файлов восходит к самым ранним дням Си и до сих пор остается наиболее распространенным. Использование модулей является новым в C++20 и дает огромные преимущества в плане гигиены кода и времени компиляции.

3.2.1 Заголовочные файлы

Традиционно мы помещаем объявления, указывающие интерфейс к фрагменту кода, который мы рассматриваем как модуль, в файл с именем, указывающим на его предполагаемое использование. Например:

```
// Vector.h:

class Vector {
public:
    Vector(int s);
    double& operator[](int i);
    int size();
private:
    double* elem;           // elem points to an array of sz doubles
    int sz;
};
```

Это объявление размещено в файле **Vector.h**. Затем пользователи **#include** этот файл, называемый *заголовочным файлом*, для доступа к этому интерфейсу. Например:

```
// user.cpp:

#include "Vector.h"           // get Vector's interface
#include <cmath>               // get the standard-library math function
                              // interface including sqrt()

double sqrt_sum(const Vector& v)
{
    double sum = 0;
    for (int i=0; i!=v.size(); ++i)
        sum+=std::sqrt(v[i]);    // sum of square roots
    return sum;
}
```

Чтобы помочь компилятору обеспечить согласованность, файл **.cpp**, предоставляющий реализацию **Vector**, также будет включать файл **.h**, предоставляющий его интерфейс:

```
// Vector.cpp:

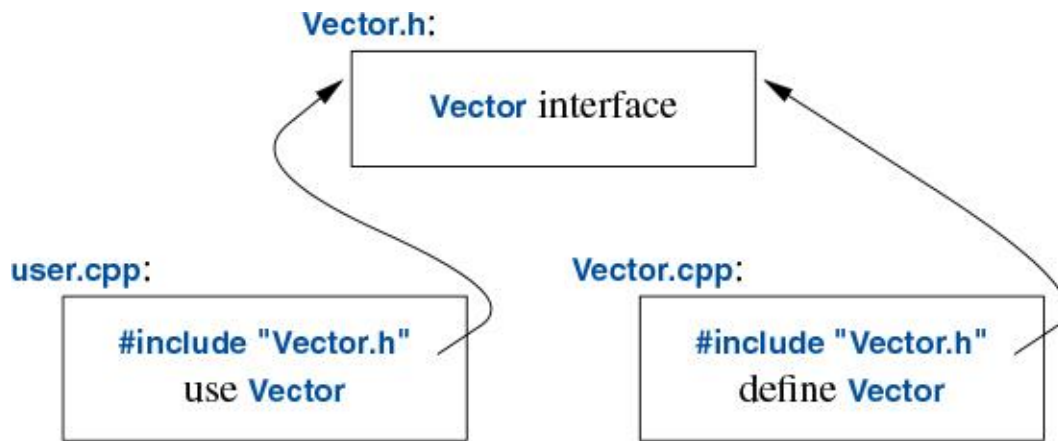
#include "Vector.h"           // get Vector's interface

Vector::Vector(int s)
    :elem(new double[s]), sz{s}    // initialize members
{
}

double& Vector::operator[](int i)
{
    return elem[i];
}

int Vector::size()
{
    return sz;
}
```

Код в **user.cpp** и **Vector.cpp** оба используют информацию об интерфейсе **Vector**, представленную в **Vector.h**, но в остальном эти два файла независимы и могут быть скомпилированы отдельно. Графически фрагменты программы можно представить следующим образом:



Наилучший подход к организации программы - представлять ее как набор модулей с четко определенными зависимостями. Заголовочные файлы представляют эту модульность с помощью файлов, а затем используют эту модульность посредством раздельной компиляции.

Файл `.cpp`, который компилируется сам по себе (включая `h`-файлы, которые он `#include`), называется *единицей трансляции*. Программа может состоять из тысяч единиц трансляции.

Использование заголовочных файлов и `#include` - это очень старый способ имитации модульности со значительными недостатками:

- *Время компиляции*: Если вы `#include header.h` в 101 единицу трансляции, текст `header.h` будет обработан компилятором 101 раз
- *Зависимость от порядка*: Если мы включаем `#include header1.h` перед `header2.h`, то объявления и макросы (§19.3.2.1) в `header1.h` могут повлиять на код в `header2.h`. Если вместо этого вы включите `#include header2.h` перед `header1.h`, то уже `header2.h` может повлиять на код в `header1.h`.
- *Несоответствия*: Определение объекта, такого как тип или функция, в одном файле, а затем определение его немного по-другому в другом файле может привести к сбоям или незначительным ошибкам. Это может произойти, если мы – случайно или намеренно – объявим объект отдельно в двух исходных файлах, вместо того, чтобы поместить его в заголовок, или из-за зависимостей от порядка включения между различными файлами заголовков.
- *Транзитивность*: весь код, необходимый для объявления в заголовочном файле, должен присутствовать в этом заголовочном файле. Это приводит к массовому раздуванию кода, поскольку заголовочные файлы `#include` в себя другие заголовки, и это приводит к тому, что пользователь заголовочного файла – случайно или намеренно – становится зависимым от таких деталей реализации.

Очевидно, что это не идеально, и этот метод был основным источником трат времени на компиляцию и ошибок с тех пор, как он был впервые введен в С в начале 1970-х годов. Однако использование заголовочных файлов было жизнеспособным на протяжении десятилетий, и старый код, использующий `#include`, будет “жить” очень долго, потому что обновление больших программ может быть дорогостоящим и отнимать много времени.

3.2.2 Модули

В C++20 у нас наконец-то появился поддерживаемый языком способ прямого выражения модульности (§19.2.4). Рассмотрим, как выразить `Vector` и `sqrt_sum()` в примере из §3.2 с использованием `module`:

```
export module Vector;      // defining the module called "Vector"

export class Vector {
public:
    Vector(int s);
    double& operator[](int i);
    int size();
private:
    double* elem;          // elem points to an array of sz doubles
    int sz;
};

Vector::Vector(int s)
    :elem{new double[s]}, sz{s}    // initialize members
{
}

double& Vector::operator[](int i)
{
    return elem[i];
}

int Vector::size()
{
    return sz;
}

export bool operator==(const Vector& v1, const Vector& v2)
{
    if (v1.size() != v2.size())
        return false;
    for (int i = 0; i < v1.size(); ++i)
        if (v1[i] != v2[i])
            return false;
    return true;
}
```

Здесь определён `module` с именем `Vector`, который экспортирует класс `Vector`, все его функции-члены и функцию, не являющуюся членом, перегружающую оператор `==`.

Способ, которым мы используем этот `module`, заключается в импорте `import` его туда, где он нам нужен. Например:

```
// file user.cpp:

import Vector;            // get Vector's interface
#include <cmath>           // get the standard-library math function interface including
                           sqrt()

double sqrt_sum(Vector& v)
{
    double sum = 0;
    for (int i=0; i!=v.size(); ++i)
        sum+=std::sqrt(v[i]);    // sum of square roots
    return sum;
}
```

Я мог бы также **import**ировать математические функции стандартной библиотеки, но я использовал старомодный **#include** просто для того, чтобы показать, что мы можем смешивать старое и новое. Такое смешивание необходимо для постепенного обновления старого кода, использующего **#include** на использование **import**.

- Модуль компилируется только один раз (а не в каждой единице трансляции, в которой он используется).
- Два модуля могут быть **import**ированы в любом порядке без изменения их значения.
- Если вы **import** или **#include** что-либо в модуль, пользователи вашего модуля неявно получают доступ к этому (и не беспокоятся об этом): **import** не является транзитивным

Влияние модулей на обслуживаемость кода и производительность во время компиляции может быть впечатляющим. Например, по моим измерениям время компиляции программы “Hello, World!”, с использованием

```
import std;
```

в 10 раз быстрее, чем версия, с использованием

```
#include<iostream>
```

Удивительный результат, несмотря на то, что модуль **std** содержит всю стандартную библиотеку, содержащую более чем в 10 раз больше информации, чем заголовок **<iostream>**. Причина в том, что модули экспортируют только интерфейсы, тогда как заголовки дает компилятору все, что он прямо или косвенно содержит. Это позволяет нам использовать большие модули, так что нам не нужно запоминать, какой именно **#include** из сбивающей с толку коллекции заголовков (§9.3.4). С этого момента я буду предполагать **import std** для всех примеров.

К сожалению, **module std** не является частью C++20. В приложении А объясняется, как получить **module std**, если реализация стандартной библиотеки еще не предоставляет его.

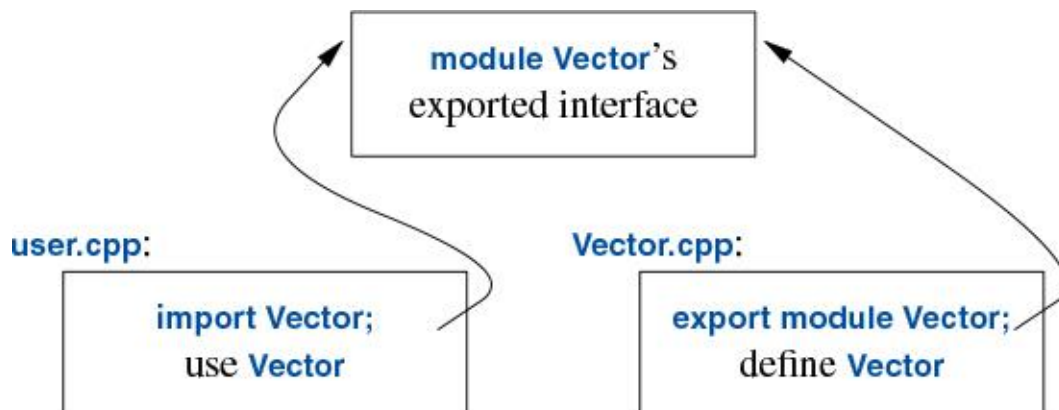
При определении модуля нам не обязательно разделять объявления и определения в отдельные файлы; мы можем, если это улучшает организацию нашего исходного кода, но это не обязательно. Мы могли бы определить простой **Vector** модуль следующим образом:

```
export module Vector;    // defining the module called "Vector"

export class Vector {
    // ...
};

export bool operator==(const Vector& v1, const Vector& v2)
{
    // ...
}
```

Графически фрагменты программы можно представить следующим образом:



Компилятор отделяет интерфейс модуля, указанный спецификаторами `export`, от деталей его реализации. Таким образом, интерфейс `Vector` генерируется компилятором и никогда явно не именуется пользователем.

Используя `module`, нам не нужно усложнять наш код, чтобы скрыть детали реализации от пользователей; `module` будет предоставлять доступ только к `экспортированным` объявлениям. Рассмотрим:

```
export module vector_printer;

import std;

export
template<typename T>
void print(std::vector<T>& v)    // this is the (only) function seen by users
{
    cout << "{\n";
    for (const T& val : v)
        std::cout << " " << val << '\n';
    cout << '}' ;
}
```

Импортируя этот тривиальный модуль, мы не получаем внезапно доступ ко всей стандартной библиотеке.

`template<typename T>` - это то, как мы параметризуем функцию с типом (§7.2).

3.3 Пространства имён

В дополнение к функциям (§1.3), классам (§2.3) и перечислениям (§2.4) C++ предлагает *пространства имен* в качестве механизма для выражения того, что некоторые объявления связаны друг с другом и что их имена не должны конфликтовать с другими именами. Например, я мог бы захотеть поэкспериментировать со своим собственным типом комплексного числа (§5.2.1, §17.4):

```
namespace My_code {
    class complex {
        // ...
    };

    complex sqrt(complex);
    // ...

    int main();
}

int My_code::main()
{
```

```

    complex z {1,2};
    auto z2 = sqrt(z);
    std::cout << '{' << z2.real() << ',' << z2.imag() << "}\n";
    // ...
}

int main()
{
    return My_code::main();
}

```

Помещая свой код в пространство имен `My_code`, я гарантирую, что мои имена не конфликтуют с именами стандартных библиотек в пространстве имен `std` (§3.3). Эта мера предосторожности разумна, поскольку стандартная библиотека действительно обеспечивает поддержку арифметики `complex` комплексных чисел (§5.2.1, §17.4).

Самый простой способ получить доступ к имени в другом пространстве имен - указать его с помощью имени пространства имён (например, `std::cout` и `My_code::main`). “Настоящий `main()`” определен в глобальном пространстве имен, то есть не является локальным для определенного пространства имен, класса или функции.

Если повторное указание имени становится утомительным или отвлекающим, мы можем поместить имя в область видимости с помощью объявления `using`:

```

void my_code(vector<int>& x, vector<int>& y)
{
    using std::swap;      // make the standard-library swap available locally
    // ...
    swap(x,y);            // std::swap()
    other::swap(x,y);     // some other swap()
    // ...
}

```

Объявление `using` делает имя из пространства имен пригодным для использования, как если бы оно было объявлено в области, в которой к нему обращаются. После `using std::swap`, можно обращаться к `swap` так как если бы он был объявлен в `my_code()`.

Чтобы получить доступ ко всем именам в пространстве имен стандартной библиотеки, мы можем использовать `using namespace`:

```
using namespace std;
```

Директива `using` делает имена из именованного пространства имен доступными из области, в которую мы поместили эту директиву. Поэтому, после директивы `using std` мы можем просто написать `cout` вместо `std::cout`. Например, мы могли бы избежать повторяющихся префиксов `std::` в нашем тривиальном модуле `vector_printer`:

```

export module vector_printer;

import std;
using namespace std;

export
template<typename T>
void print(vector<T>& v) // this is the (only) function seen by users
{
    cout << "{\n";
    for (const T& val : v)
        cout << " " << val << '\n';
    cout << '}' ;
}

```


Важно отметить, что использование директивы `namespace` не влияет на пользователей наших модулей; это деталь реализации, локальная для модуля.

Используя директиву `using`, мы теряем возможность выборочно использовать имена из этого пространства имен, поэтому это средство следует использовать осторожно, обычно для библиотеки, которая широко распространена в приложении (например, `std`), или во время перехода для приложения, которое не использовало пространства имен.

Пространства имен в основном используются для организации более крупных программных компонентов, таких как библиотеки. Они упрощают составление программы из отдельно разработанных частей.

3.4 Аргументы функции и возвращаемые значения

Основным и рекомендуемым способом передачи информации из одной части программы в другую является вызов функции. Информация, необходимая для выполнения задачи, передается в качестве аргументов функции, а полученные результаты передаются обратно в виде возвращаемых значений. Например:

```
int sum(const vector<int>& v)
{
    int s = 0;
    for (const int i : v)
        s += i;
    return s;
}

vector fib = {1, 2, 3, 5, 8, 13, 21};

int x = sum(fib);           // x becomes 53
```

Существуют и другие пути передачи информации между функциями, такие как глобальные переменные (§1.5) и общее состояние в объекте класса (глава 5). Глобальные переменные настоятельно не рекомендуются как известный источник ошибок, а общее состояние, как правило, должно использоваться только функциями, совместно реализующими четко определенную абстракцию (например, функции-члены класса; §2.3).

Учитывая важность передачи информации в функции и из них, неудивительно, что существует множество способов сделать это. Ключевыми проблемами являются:

- Является ли объект копируемым или используется совместно?
- Если объект используется совместно, может ли он изменяться?
- Если объект перемещается, остаётся ли после этого “пустой объект” (§6.2.2)?

По умолчанию как при передаче аргументов, так и при возврате значения - “создаётся копия” (§1.9), но многие копии могут быть неявно оптимизированы для перемещения.

В примере `sum()` результирующий `int` копируется из `sum()`, но было бы неэффективно и бессмысленно копировать потенциально очень большой `vector` в `sum()`, поэтому аргумент передается по ссылке (обозначается символом `&`; §1.7).

У `sum()` нет причин изменять свой аргумент. Эта неизменяемость указывается объявлением аргумента `vector` как `const` (§1.6), поэтому `vector` передается по `const`-ссылке.

3.4.1 Передача аргументов

Сначала рассмотрим, как передать значения в функцию. По умолчанию мы копируем (“передаём по значению”), но, если мы хотим сослаться на объект в среде вызывающего объекта, мы используем ссылку (“передаём по ссылке”). Например:

```
void test(vector<int> v, vector<int>& rv)    // v is passed by value;
{                                           // rv is passed by reference
    v[1] = 99;                            // modify v (a local variable)
    rv[2] = 66;                          // modify whatever rv refers to
}

int main()
{
    vector fib = {1, 2, 3, 5, 8, 13, 21};
    test(fib, fib);
    cout << fib[1] << ' ' << fib[2] << '\n';    // prints 2 66
}
```

Когда мы заботимся о производительности, мы обычно передаем малые значения по значению, а большие - по ссылке. Здесь “малые” означает “что-то, что действительно дешево скопировать”. Что именно означает “малый”, зависит от архитектуры машины; “размер двух-трех указателей или меньше” - хорошее эмпирическое правило. Если для вас важна производительность, измерьте разные варианты и выберите лучший.

Если мы хотим передать аргументы по ссылке из соображений производительности, но нам не нужно их изменять, мы передаем по `const`-ссылке, как в примере `sum()`. Это, безусловно, самый распространенный случай в обычном хорошем коде: он быстр и не подвержен ошибкам.

Нередко аргумент функции имеет значение по умолчанию, то есть значение, которое считается предпочтительным или просто наиболее распространенным. Мы можем указать такое значение с помощью *аргумента функции по умолчанию*. Например:

```
void print(int value, int base = 10);    // print value in base "base"

print(x, 16);                          // hexadecimal
print(x, 60);                          // sexagesimal (Sumerian)
print(x);                              // use the default: decimal
```

Это более простая в обозначениях альтернатива перегрузке:

```
void print(int value, int base);        // print value in base "base"

void print(int value)                  // print value in base 10
{
    print(value, 10);
}
```

Использование аргументов по умолчанию означает, что существует только одно определение функции. Обычно это хорошо для понимания и размера кода. Когда нам нужен разный код для реализации одной и той же семантики для разных типов, то лучше использовать перегрузку.

3.4.2 Возвращение значений

Как только мы вычислили результат, нам нужно извлечь его из функции и вернуть вызывающему. Опять же, по умолчанию для возврата значения используется копирование, и для небольших объектов это идеально. Мы возвращаем “по ссылке” только то-

гда, когда хотим предоставить вызывающему доступ к чему-то, что не является локальным для функции. Например, **Vector** может предоставить пользователю доступ к элементу посредством оператора индекса:

```
class Vector {
public:
    // ...
    double& operator[](int i) { return elem[i]; }    // return reference
                                                    // to ith element
private:
    double* elem;    // elem points to an array of sz
    // ...
};
```

i-й элемент **Vector** существует независимо от вызова оператора индекса, поэтому мы можем вернуть ссылку на него.

С другой стороны, локальная переменная исчезает при возврате функции, поэтому мы не должны возвращать указатель или ссылку на нее:

```
int& bad()
{
    int x;
    // ...
    return x;    // bad: return a reference to the local variable x
}
```

К счастью, все основные компиляторы C++ обнаружат очевидную ошибку в **bad()**.

Возврат ссылки или значения “малого” типа эффективен, но как передавать большие объемы информации из функции? Рассмотрим:

```
Matrix operator+(const Matrix& x, const Matrix& y)
{
    Matrix res;
    // ... for all res[i,j], res[i,j] = x[i,j]+y[i,j] ...
    return res;
}

Matrix m1, m2;
// ...
Matrix m3 = m1+m2;    // no copy
```

Matrix может быть *очень* большой и дорогостоящей для копирования даже на современном аппаратном обеспечении. Поэтому чтобы избежать копирования, мы даем **Matrix** конструктор перемещения (§6.2.2), который очень дешево возвращает **Matrix** из **operator+()**. Даже если мы не определяем конструктор перемещения, компилятор часто способен оптимизировать копирование (исключить копирование) и построить **Matrix** именно там, где это необходимо. Это называется *пропуском копирования (copy elision)*.

Мы *не* должны скатываться к ручному управлению памятью:

```
Matrix* add(const Matrix& x, const Matrix& y)    // complicated and error-prone
{
    Matrix* p = new Matrix;
    // ... for all *p[i,j], *p[i,j] = x[i,j]+y[i,j] ...
    return p;
}

Matrix m1, m2;
// ...
Matrix* m3 = add(m1,m2);    // just copy a pointer
```

```
// ...  
delete m3; // easily forgotten
```

К сожалению, возврат большого объекта путем передачи указателя на него является распространенным явлением в старом коде и основным источником трудноисправимых ошибок. Не пишите такой код! `Matrix operator+()` по крайней мере так же эффективен, как `Matrix add()`, но гораздо проще в определении, проще в использовании и менее подвержен ошибкам.

Если функция не может выполнить требуемую от нее задачу, она может выдать исключение (§4.2). Это может помочь избежать засорения кода проверками кодов ошибок для выявления “исключительных проблем”.

3.4.3 Выведение типа возвращаемого значения

Тип возвращаемого значения функции может быть выведен из ее возвращаемого значения. Например:

```
auto mul(int i, double d) { return i*d; } // here, "auto" means "deduce  
// the return type"
```

Это может быть удобно, особенно для универсальных функций (§7.3.1) и лямбд (§7.3.3), но использовать их следует осторожно, поскольку выведенный тип не обеспечивает стабильного интерфейса: изменение реализации функции (или лямбда-выражения) может изменить ее тип.

3.4.4 Суффиксная запись типа возвращаемого значения

Почему возвращаемый тип указывается перед именем и аргументами функции? Причина в основном историческая. Именно так это делали Fortran, C и Simula (и делают до сих пор). Однако иногда нам нужно посмотреть на аргументы, чтобы определить тип результата. Выведение возвращаемого типа - один из примеров этого, но не единственный. В примерах, выходящих за рамки этой книги, проблема возникает в связи с пространствами имен (§3.3), лямбдами (§7.3.3) и концептами (§8.2). Следовательно, мы разрешаем добавлять возвращаемый тип после списка аргументов, где мы хотим четко указать возвращаемый тип. Это делает `auto` означаящим “возвращаемый тип будет упомянут позже или будет выведен”. Например:

```
auto mul(int i, double d) -> double { return i*d; } // the return type is "double"
```

Как и в случае с переменными (§1.4.2), мы можем использовать это обозначение для более аккуратного выстраивания имен. Сравните это использование суффиксной записи возвращаемых типов с версией в (§1.3):

```
auto next_elem() -> Elem*;  
auto exit(int) -> void;  
auto sqrt(double) -> double;
```

Я нахожу суффиксную запись возвращаемого типа с более логичной, чем традиционную префиксную, но поскольку подавляющее большинство кода использует традиционное обозначение, я придерживаюсь его в этой книге.

3.4.5 Структурное связывание

Функция может возвращать только одно значение, но это значение может быть объектом класса со многими членами. Это позволяет нам элегантно возвращать множество значений. Например:

```
struct Entry {
    string name;
    int value;
};

Entry read_entry(istream& is)    // naive read function (for a better
{                                // version, see §11.5)
    string s;
    int i;
    is >> s >> i;
    return {s,i};
}

auto e = read_entry(cin);

cout << "{ " << e.name << " , " << e.value << " }\n";
```

Здесь `{s,i}` используется для построения возвращаемого значения типа `Entry`. Аналогично, мы можем “распаковать” члены `Entry` в локальные переменные:

```
auto [n,v] = read_entry(is);
cout << "{ " << n << " , " << v << " }\n";
```

Конструкция `auto [n,v]` объявляет две локальные переменные `n` и `v` с типами, выведенными из возвращаемого `read_entry()` типа. Этот механизм присвоения локальных имен членам объекта класса называется *структурным связыванием*.

Рассмотрим другой пример:

```
map<string,int> m;
// ... fill m ...
for (const auto [key,value] : m)
    cout << "{ " << key << " , " << value << " }\n";
```

Как обычно, мы можем украсить `auto` с помощью `const` и `&`. Например:

```
void incr(map<string,int>& m)    // increment the value of each element of m
{
    for (auto& [key, value] : m)
        ++value;
}
```

Когда структурное связывание используется для класса без приватных полей, легко увидеть, как выполняется связывание: для связывания должно быть определено столько же имен, сколько элементов данных в объекте класса, и каждое имя, введенное в связывание, называет соответствующий элемент. Не будет никакой разницы в качестве объектного кода по сравнению с явным использованием составного объекта. В частности, использование структурного связывания не подразумевает копирование `struct`. Кроме того, возврат простого `struct` редко включает в себя копирование, поскольку простые возвращаемые типы могут быть созданы непосредственно в том месте, где они необходимы (§3.4.2). Использование структурного связывания - это все о том, как наилучшим образом выразить идею.

Также возможно обрабатывать классы, доступ к которым осуществляется через функции-члены. Например:

```
complex<double> z = {1,2};  
auto [re,im] = z+2;           // re=3; im=2
```

`complex` имеет два элемента данных, но его интерфейс состоит из функций доступа, таких как `real()` и `imag()`. Сопоставление `complex<double>` двум локальным переменным, таким как `re` и `im`, осуществимо и эффективно, но методика выполнения этого выходит за рамки данной книги.

3.5 Советы

- [1] Различайте объявления (используемые в качестве интерфейсов) и определения (используемые в качестве реализаций); §3.1.
- [2] Отдавайте предпочтение `module`, а не заголовкам (где `module` поддерживаются); §3.2.2.
- [3] Используйте заголовочные файлы для представления интерфейсов и подчеркивания логической структуры; §3.2; [CG: SF.3].
- [4] `#include` заголовков в исходный файл, который реализует его функции; §3.2; [CG: SF.5].
- [5] Избегайте определений не-`inline` функций в заголовочных файлах; §3.2; [CG: SF.2].
- [6] Используйте пространства имен для выражения логической структуры; §3.3; [CG: SF.20].
- [7] Используйте директивы `using` для перехода, для базовых библиотек (таких как `std`) или в пределах локальной области видимости; §3.3; [CG: SF.6] [CG: SF.7].
- [8] Не помещайте директиву `using` в заголовочный файл; §3.3; [CG: SF.7].
- [9] Передавайте “малые” значения по значению, а “большие” значения по ссылке; §3.4.1; [CG: F.16].
- [10] Предпочитайте передачу по `const`-ссылке обычной передаче по ссылке; §3.4.1; [CG: F.17].
- [11] Возвращайте информацию как значения, возвращаемые функцией (а не с помощью выходных параметров); §3.4.2; [CG: F.20] [CG: F.21].
- [12] Не злоупотребляйте выводением типа возвращаемого значения; §3.4.2.
- [13] Не злоупотребляйте структурным связыванием; явное указание возвращаемого типа часто делает код более читаемым; §3.4.5.

Обработка ошибок

Не перебивай меня, пока я перебиваю.

– Winston S. Churchill

- [Введение](#)
- [Прерывания](#)
- [Инварианты](#)
- [Альтернативные способы обработки ошибок](#)
- [Утверждения](#)

[assert\(\)](#); [static_assert\(\)](#); [noexcept](#)

- [Советы](#)

4.1 Введение

Обработка ошибок - это большая и сложная тема с проблемами и ответвлениями, которые выходят далеко за рамки возможностей языка программирования и касаются методов и инструментов программирования. Однако C++ предоставляет несколько полезных функций. Основным инструментом является сама система типов. Вместо того, чтобы кропотливо создавать наши приложения на основе встроенных типов (например, `char`, `int` или `double`) и операторов (например, `if`, `while` и `for`), мы создаем типы (например, `string`, `map` и `thread`) и алгоритмы (например, `sort()`, `find_if()` и `draw_all()`), которые подходят для наших приложений. Такие высокоуровневые конструкции упрощают наше программирование, ограничивают возможности для ошибок (например, вы вряд ли попытаетесь применить обход дерева к диалоговому окну) и увеличивают шансы компилятора на обнаружение ошибок. Большинство конструкций языка C++ предназначены для проектирования и реализации элегантных и эффективных абстракций (например, пользовательских типов и алгоритмов, использующих их). Одним из эффектов использования таких абстракций является то, что точка, в которой может быть обнаружена ошибка времени выполнения, отделена от точки, в которой она может быть обработана. По мере роста программ, и особенно при интенсивном использовании библиотек, стандарты обработки ошибок становятся важными. Это хорошая идея - сформулировать стратегию обработки ошибок на ранних стадиях разработки программы.

4.2 Исключения

Рассмотрим еще раз пример `Vector`. Что *следует* делать, когда мы пытаемся получить доступ к элементу, который находится вне диапазона для вектора из §2.3?

- Автор `Vector` не знает, что пользователь хотел бы сделать в этом случае (автор `Vector` обычно даже не знает, в какой программе будет работать `Vector`).
- Пользователь `Vector` не может последовательно обнаружить проблему (если бы пользователь мог, доступ за пределы диапазона не произошел бы в первую очередь)

Предполагая, что доступ за пределы диапазона - это своего рода ошибка, от которой мы хотим избавиться, решение заключается в том, чтобы разработчик `Vector` обнаружил попытку доступа за пределы диапазона и сообщил об этом пользователю. Затем пользователь может предпринять соответствующие действия. Например, `Vector::operator[]()` может обнаружить попытку доступа вне диапазона и выдать исключение `out_of_range`:

```
double& Vector::operator[](int i)
{
    if (!(0<i && i<size()))
        throw out_of_range{"Vector::operator[]"};
    return elem[i];
}
```

`throw` передает управление обработчику исключений типа `out_of_range` в некоторой функции, которая прямо или косвенно вызывает `Vector::operator[]()`. Чтобы сделать это, реализация *разворачивает* стек вызовов функций по мере необходимости, чтобы вернуться к контексту вызывающего объекта. То есть механизм обработки исключений будет выходить из областей видимости и функций по мере необходимости, чтобы вернуться к вызывающей стороне, которая выразила заинтересованность в обработке такого рода исключений, вызывая деструкторы (§5.2.2) по мере необходимости. Например:

```
void f(Vector& v)
{
    // ...
    try { // out_of_range exceptions thrown in this block are handled by the handler
        defined below
        compute1(v);           // might try to access beyond the end of v
        Vector v2 = compute2(v); // might try to access beyond the end of v
        compute3(v2);          // might try to access beyond the end of v2
    }
    catch (const out_of_range& err) { // oops: out_of_range error
        // ... handle range error ...
        cerr << err.what() << '\n';
    }
    // ...
}
```

Мы помещаем код, для которого нас интересует обработка исключений, в блок `try`. Вызовы `compute1()`, `compute2()`, и `compute3()` предназначены для представления кода, для которого сложно заранее определить, произойдет ли ошибка диапазона. Предложение `catch` предоставляется для обработки исключений типа `out_of_range`. Если бы `f()` не был подходящим местом для обработки таких исключений, мы бы не использовали блок `try`, а вместо этого позволили исключению неявно передаваться вызывающему `f()`.

Тип `out_of_range` определен в стандартной библиотеке (в `<stdexcept>`) и фактически использует некоторыми функциями доступа к контейнерам стандартной библиотеки.

Я перехватил исключение по ссылке, чтобы избежать копирования, и использовал функцию `what()`, чтобы вывести сообщение об ошибке, полученное в момент `throw`.

Использование механизмов обработки исключений может сделать обработку ошибок более простой, систематизированной и читабельной. Чтобы достичь этого, не злоупотребляйте операторами `try`. Во многих программах обычно существуют десятки вызовов функций между `throw` и функцией, которая может разумно обработать выданное исключение. Таким образом, большинство функций должны просто разрешать распространение исключения вверх по стеку вызовов. Основной метод, позволяющий упростить и систематизировать обработку ошибок (называемый *получением ресурсов при инициализации*; RAII), объясняется в §5.2.2. Основная идея RAII заключается в том, чтобы конструктор получал ресурсы, необходимые для работы класса, а деструктор освобождал все полученные ресурсы, таким образом делая освобождение ресурсов гарантированным и неявным.

4.3 Инварианты

Использование исключений для сигнализации о доступе за пределы диапазона является примером того, как функция проверяет свой аргумент и отказывается работать, когда не выполняется *предусловие*. Если бы мы формально указали оператор индекса `Vector`, мы бы сказали что-то вроде “индекс должен находиться в диапазоне `[0 : size())`”, и это было фактически то, что мы проверили в нашем `operator[]()`. Обозначение `[a:b)` обозначает полуоткрытый диапазон, в котором `a` является частью диапазона, а `b` - нет. Всякий раз, когда мы определяем функцию, мы должны учитывать, каковы ее предварительные условия, и решить, стоит ли их проверять (§4.4). Для большинства приложений хорошей идеей является проверка простых инвариантов; см. также §4.5.

Однако `operator[]()` работает с объектами типа `Vector`, и ничто из того, что он делает, не имеет смысла, если члены `Vector` не имеют “разумных” значений. В частности, мы действительно сказали “`elem` указывает на массив из `sz` элементов типа `double`”, но мы сказали это только в комментарии. Такое утверждение о том, что предполагается истинным для класса, называется *инвариантом* класса, или просто *инвариантом*. Задача конструктора - установить инвариант для своего класса (чтобы функции-члены могли полагаться на него), а функций-членов - гарантировать, что инвариант сохраняется при выходе. К сожалению, наш конструктор `Vector` лишь частично выполнил свою работу. Он правильно инициализировал элементы `Vector`, но не смог проверить, что переданные ему аргументы имеют смысл. Рассмотрим:

```
Vector v(-27);
```

Это, вероятно, вызовет хаос.

Вот более подходящее определение:

```
Vector::Vector(int s)
{
    if (s<0)
        throw length_error{"Vector constructor: negative size"};
    elem = new double[s];
    sz = s;
}
```

Я использую исключение стандартной библиотеки `length_error` для сообщения об отрицательном количестве элементов, потому что некоторые операции стандартной библиотеки используют это исключение для сообщения о проблемах такого рода. Если

operator `new` не может выделить необходимую память, он бросает исключение `std::bad_alloc`. Теперь мы можем написать:

```
void test(int n)
{
    try {
        Vector v(n);
    }
    catch (std::length_error& err) {
        // ... handle negative size ...
    }
    catch (std::bad_alloc& err) {
        // ... handle memory exhaustion ...
    }
}

void run()
{
    test(-27);           // throws length_error (-27 is too small)
    test(1 000 000 000); // may throw bad_alloc
    test(10);           // likely OK
}
```

Исчерпание памяти происходит, если вы запрашиваете больше памяти, чем предлагает компьютер, или если ваша программа уже почти потребила лимит, и ваш запрос превышает лимит. Обратите внимание, что современные операционные системы обычно предоставляют вам больше места, чем поместится в физической памяти, поэтому запрос слишком большого объема памяти может привести к серьезному замедлению работы задолго до запуска `bad_alloc`.

Вы можете определить свои собственные классы для использования в качестве исключений и сделать так, чтобы они несли столько информации, сколько вам нужно, от момента обнаружения ошибки до момента, когда она может быть обработана (§4.2). Нет необходимости использовать иерархию исключений стандартной библиотеки.

Часто функция не имеет возможности выполнить назначенную ей задачу после возникновения исключения. Затем “обработка” исключения означает выполнение некоторой минимальной локальной очистки и повторное создание исключения. Например:

```
void test(int n)
{
    try {
        Vector v(n);
    }
    catch (std::length_error&) { // do something and rethrow
        cerr << "test failed: length error\n";
        throw; // rethrow
    }
    catch (std::bad_alloc&) { // ouch! this program is not designed to
                             // handle memory exhaustion
        std::terminate(); // terminate the program
    }
}
```

В хорошо продуманном коде блоки `try` встречаются редко. Избегайте чрезмерного использования, систематически используйте RAII (§5.2.2, §6.3).

Понятие инвариантов занимает центральное место при проектировании классов, а предварительные условия играют аналогичную роль при проектировании функций:

- Формулировка инвариантов помогает нам точнее понять, чего мы хотим.

- Инварианты вынуждают нас быть конкретными; это дает нам больше шансов написать корректный код

Понятие инвариантов лежит в основе представлений C++ об управлении ресурсами, поддерживаемых конструкторами ([глава 5](#)) и деструкторами (§5.2.2, §15.2.1).

4.4 Альтернативные способы обработки ошибок

Обработка ошибок является серьезной проблемой во всех реальных программах, поэтому, естественно, существует множество подходов. Если обнаружена ошибка, и она не может быть обработана локально в функции, функция должна каким-то образом сообщить о проблеме какому-либо вызывающему объекту. Создание исключения это главный механизм в C++ для этого.

Существуют языки, в которых исключения предназначены просто для того, чтобы обеспечить альтернативный механизм возврата значений. C++ не является таким языком: исключения предназначены для сообщения о невозможности выполнения данной задачи. Исключения интегрированы с конструкторами и деструкторами, чтобы обеспечить согласованную структуру для обработки ошибок и управления ресурсами (§5.2.2, §6.3). Компиляторы оптимизированы таким образом, чтобы сделать возврат значения намного дешевле, чем выбрасывание того же значения в качестве исключения.

Создание исключения - не единственный способ сообщить об ошибке, которая не может быть обработана локально. Функция может указать на то, что она не может выполнить возложенную на нее задачу с помощью:

- выбрасывания исключения
- возвращения значения, указывающего на ошибку
- завершения работы программы (путем вызова такой функции, как `terminate()`, `exit()` или `abort()` (§16.8)).

Мы возвращаем индикатор ошибки (“код ошибки”), когда:

- Сбой является нормальным и ожидаемым. Например, вполне нормально, что запрос на открытие файла завершается ошибкой (возможно, файла с таким именем не существует или, возможно, файл не может быть открыт с запрошенными правами доступа)
- Ожидается, что непосредственный вызывающий справится с обработкой ошибки.
- Произошла ошибка в одной из множества параллельных задач, и нам нужно знать, какая задача завершилась неудачей.
- В системе так мало памяти, что поддержка исключений во время выполнения вытеснила бы необходимую функциональность.

Мы используем исключение, когда:

- Ошибка настолько редка, что программист, скорее всего, забудет ее проверить. Например, когда вы в последний раз проверяли возвращаемое значение `printf()`?
- Ошибка не может быть обработана непосредственно вызывающей функцией. Вместо этого ошибка должна распространяться обратно по цепочке вызовов к “конечному вызывающему”. Например, невозможно, чтобы каждая функция в приложении надежно обрабатывала каждый сбой аллокации и отключение сети. Повторная проверка кода ошибки была бы утомительной, дорогостоящей и подверженной ошибкам. Тесты на наличие ошибок и

передача кодов ошибок в качестве возвращаемых значений могут легко затенить основную логику функции.

- В нижестоящие модули приложения могут быть добавлены новые виды ошибок, так что модули более высокого уровня не будут написаны для устранения таких ошибок. Например, когда ранее однопоточное приложение модифицируется для использования нескольких потоков, или ресурсы размещаются удаленно для доступа по сети.
- Не доступен подходящий путь возврата для кодов ошибок. Например, у конструктора нет возвращаемого значения для проверки “вызывающим”. В частности, конструкторы могут быть вызваны для нескольких локальных переменных или в частично сконструированном сложном объекте, так что очистка на основе кодов ошибок была бы довольно сложной. Аналогично, операторы обычно не имеют очевидного пути возврата для кодов ошибок. Например, `a*b+c/d`.
- Путь возврата функции усложняется или удорожается из-за необходимости передавать обратно как значение, так и индикатор ошибки (например, `pair`; §15.3.3), что может привести к использованию внешних параметров, нелокальных индикаторов состояния ошибки или других обходных путей.
- Восстановление после ошибок зависит от результатов нескольких вызовов функций, что приводит к необходимости поддерживать локальное состояние между вызовами и сложным управляющим структурам.
- Функция, обнаружившая ошибку, была обратным вызовом callback (функциональный аргумент), поэтому непосредственный вызывающий может даже не знать, какая функция была вызвана.
- Ошибка подразумевает, что требуется какое-то “отмены действий” (§5.2.2)

Мы прекращаем работу программы, когда:

- Восстановление после ошибки такого рода невозможно. Например, для многих – но не для всех систем не существует разумного способа восстановления после исчерпания памяти.
- В системах, в которых обработка ошибок основана на перезапуске потока, процесса или компьютера при обнаружении нетривиальной ошибки.

Один из способов обеспечить завершение работы приложения - добавить `noexcept` (§4.5.3) к функции, чтобы `throw` из любого места реализации функции превращался в `terminate()`. Обратите внимание, что существуют приложения, для которых безусловное завершение программы не приемлемо, поэтому необходимо использовать альтернативные варианты. Библиотека, предназначенная для общего использования никогда не должна завершаться без сообщений об ошибках.

К сожалению, эти условия не всегда логически разделимы и просты в применении. Имеют значение размер и сложность программы. Иногда по мере развития приложения приоритеты и компромиссы меняются. Требуется опыт для принятия верных решений. Если вы сомневаетесь, отдавайте предпочтение исключениям, потому что их использование лучше масштабируется и не требует внешних инструментов для проверки того, что все ошибки обработаны.

Не верьте, что все коды ошибок или все исключения являются плохими; есть очевидное применение и тому, и другому. Кроме того, не верьте мифу о том, что обработка исключений происходит медленно; зачастую она быстрее, чем правильная обработка сложных или редких ошибок и повторных проверок кодов ошибок.

RAII (§5.2.2, §6.3) необходим для простой и эффективной обработки ошибок с использованием исключений. Код, усеянный блоками `try`, часто просто отражает худшие аспекты стратегий обработки ошибок, разработанных для кодов ошибок.

4.5 Утверждения

В настоящее время не существует общего и стандартного способа написания дополнительных тестов времени выполнения для инвариантов, предусловий и т.д. Однако для многих крупных программ существует необходимость в поддержке пользователей, которые хотят полагаться на обширные run-time проверки во время тестирования, но затем разворачивать код с минимальными проверками.

Пока нам приходится полагаться на специальные механизмы. Таких механизмов существует множество. Они должны быть гибкими, общими и не предполагать никаких затрат, если они не включены. Это подразумевает простоту концепции и изощренность в реализации. Вот схема, которую я использовал:

```
enum class Error_action { ignore, throwing, terminating, logging }; // error-handling alternatives

constexpr Error_action default_Error_action = Error_action::throwing; // a default

enum class Error_code { range_error, length_error }; // individual errors

string error_code_name[] { "range error", "length error" }; // names of individual errors

template<Error_action action = default_Error_action, class C>
constexpr void expect(C cond, Error_code x) // take "action" if the expected condition "cond" doesn't hold
{
    if constexpr (action == Error_action::logging)
        if (!cond()) std::cerr << "expect() failure: " << int(x) << ' ' << error_code_name[int(x)] << '\n';
    if constexpr (action == Error_action::throwing)
        if (!cond()) throw x;
    if constexpr (action == Error_action::terminating)
        if (!cond()) terminate();
    // or no action
}
```

На первый взгляд это может показаться ошеломляющим, поскольку многие используемые языковые функции еще не представлены. Однако, по мере необходимости, он одновременно очень гибок и тривиален в использовании. Например:

```
double& Vector::operator[](int i)
{
    expect([i,this] { return 0<=i && i<size(); }, Error_code::range_error);
    return elem[i];
}
```

Здесь проверяется, находится ли индекс в диапазоне, и выполняет действие по умолчанию, вызывая исключение, если это не так. Ожидаемое выполнение условия, `0<=i&&i<size()`, передается в `expect()` как лямбда-выражение, `[i,this]{return 0<=i&&i<size();}` (§7.3.3). Проверка `if constexpr` выполняется во время компиляции (§7.4.3), поэтому для каждого вызова `expect()` выполняется не более одного теста во время выполнения. Установите `action` в значение `Error_action::ignore`, и никаких действий не будет предпринято, и никакой код не будет сгенерирован для `expect()`.

Установив `default_Error_action`, пользователь может выбрать действие, подходящее для конкретного разворачивания программы, например, завершение работы `terminating` или ведение журнала `logging`. Для поддержки логирования необходимо определить таблицу `error_code_name`. Информация о протоколировании может быть улучшена с помощью `source_location` (§16.5).

Во многих системах важно, чтобы механизм утверждений, такой как `expect()`, представлял единую точку контроля значений проверки утверждений. Поиск в большой базе кода из инструкций `if`, которые на самом деле являются проверками предположений, обычно непрактичен.

4.5.1 `assert()`

Стандартная библиотека предлагает отладочный макрос `assert()`, для утверждений, что условие должно выполняться во время выполнения программы. Например:

```
void f(const char* p)
{
    assert(p!=nullptr);    // p must not be the nullptr
    // ...
}
```

Если условие `assert()` не выполняется в “режиме отладки”, программа завершается. Если программа скомпилирована не в режиме отладки, `assert()` не проверяется. Это довольно грубо и негибко, но часто лучше, чем ничего.

4.5.2 `static_assert()`

Исключения сообщают об ошибках, обнаруженных во время выполнения. Обычно предпочтительнее, если ошибка может быть обнаружена во время компиляции. Именно для этого предназначена большая часть системы типов и средств для указания интерфейсов к пользовательским типам. Однако мы также можем выполнять простые проверки большинства свойств, которые известны во время компиляции, и сообщать о сбоях, в виде сообщений компилятора об ошибках. Например:

```
static_assert(4<=sizeof(int), "integers are too small"); // check integer size
```

Это приведет к выводу `integers are too small`, если `4<=sizeof(int)` не выполняется; то есть, если `int` в этой системе не содержит по крайней мере 4 байт. Мы называем такие заявления об ожиданиях утверждениями.

Механизм `static_assert` может использоваться для всего, что может быть выражено в терминах константных выражений (§1.6). Например:

```
constexpr double C = 299792.458;           // km/s

void f(double speed)
{
    constexpr double local_max = 160.0/(60*60);           // 160 km/h == 160.0/(60*60)
    km/s

    static_assert(speed<C,"can't go that fast");           // error: speed must be a constant
    static_assert(local_max<C,"can't go that fast");       // OK
    // ...
}
```


В общем случае `static_assert(A,S)` выводит **S** как сообщение компилятора об ошибке, если **A** не **true**. Если вы не хотите, чтобы печаталось конкретное сообщение, опустите **S**, и компилятор выдаст сообщение по умолчанию:

```
static_assert(4<=sizeof(int));           // use default message
```

Сообщение по умолчанию обычно представляет собой местоположение `static_assert` в исходниках и символьное представление утвержденного предиката.

Одним из важных применений `static_assert` является создание утверждений о типах, используемых в качестве параметров в обобщённом программировании (§8.2, §16.4).

4.5.3 noexcept

Функция, которая никогда не должна выдавать исключение, может быть объявлена **noexcept**. Например:

```
void user(int sz) noexcept
{
    Vector v(sz);
    iota(&v[0],&v[sz],1);    // fill v with 1,2,3,4... (see §17.3)
    // ...
}
```

Если все благие намерения и планирование терпят неудачу, так что `user()` по-прежнему выдает ошибку, вызывается `std::terminate()` для немедленного завершения программы.

Бездумное разбрызгивание **noexcept** на функции опасно. Если функция **noexcept** вызывает функцию, которая выдает исключение, ожидая, что оно будет перехвачено и обработано, **noexcept** превращает это в фатальную ошибку. Кроме того, **noexcept** вынуждает автора обрабатывать ошибки с помощью некоторой формы кодов ошибок, которые могут быть сложными, подверженными ошибкам и дорогостоящими (§4.4). Как и другие мощные языковые функции, **noexcept** следует применять с пониманием и осторожностью.

4.6 Советы

- [1] Бросайте исключение, чтобы указать, что вы не можете выполнить порученную задачу; §4.4; [CG: E.2].
- [2] Используйте исключения только для обработки ошибок; §4.4; [CG: E.3].
- [3] Невозможность открыть файл или достичь конца итерации являются ожидаемыми событиями, а не исключительными; §4.4.
- [4] Используйте коды ошибок, когда ожидается, что непосредственный вызывающий обработает ошибку; §4.4.
- [5] Бросайте исключение для ошибок, которые, как ожидается, будут передаваться в цепочке вызовов функций; §4.4.
- [6] Если вы сомневаетесь, использовать ли исключение или код ошибки, отдайте предпочтение исключениям; §4.4.
- [7] Разработайте стратегию обработки ошибок на ранних стадиях проектирования; §4.4; [CG: E.12].
- [8] Используйте специально разработанные пользовательские типы в качестве исключений (не встроенные типы); §4.2.

- [9] Не пытайтесь перехватывать каждое исключение в каждой функции; §4.4; [CG: E.7].
- [10] Вам не обязательно использовать иерархию классов исключений стандартной библиотеки; §4.3.
- [11] Предпочтительнее использование RAII явным блокам `try`; §4.2, §4.3; [CG: E.6].
- [12] Пусть конструктор устанавливает инвариант и бросает исключение, если он не может; §4.3; [CG: E.5].
- [13] Разработайте свою стратегию обработки ошибок на основе инвариантов; §4.3; [CG: E.4].
- [14] То, что можно проверить во время компиляции, обычно лучше всего проверять во время компиляции; §4.5.2 [CG: P.4] [CG: P.5].
- [15] Используйте механизм утверждения, чтобы обеспечить единую точку контроля обработкой ошибок; §4.5.
- [16] Концепты (§8.2) являются предикатами времени компиляции и поэтому часто полезны в утверждениях; §4.5.2.
- [17] Если ваша функция может не бросать исключения, объявите её `noexcept`; §4.4; [CG: E.12].
- [18] Не используйте `noexcept` бездумно; §4.5.3.

Классы

*Эти типы не являются “абстрактными”, они такие же реальные, как `int` и `float`.
– Doug McIlroy*

- [Введение](#)

[Классы](#)

- [Конкретные типы](#)

[Арифметические типы](#); [Контейнер](#); [Инициализация контейнеров](#)

- [Абстрактные типы](#)
- [Виртуальные функции](#)
- [Иерархия классов](#)

[Преимущества иерархий](#); [Навигация по иерархии](#); [Предотвращение утечки ресурсов](#)

- [Советы](#)

5.1 Введение

Цель этой и трех следующих глав - дать вам представление о поддержке языком C++ абстракций и управления ресурсами, не вдаваясь в подробности:

- В этой главе неформально представлены способы определения и использования новых типов (*пользовательских типов*). В частности, в нем представлены основные свойства, методы реализации и языковые средства, используемые для *конкретных классов, абстрактных классов и иерархий классов*.
- В [главе 6](#) представлены операции, которые имеют определенное значение в C++, такие как конструкторы, деструкторы и присваивания. В нем излагаются правила их комбинированного использования для управления жизненным циклом объектов и поддержки простого, эффективного и полного управления ресурсами.
- В [главе 7](#) представлены шаблоны как механизм параметризации типов и алгоритмов с помощью других типов и алгоритмов. Вычисления для пользовательских и встроенных типов представлены в виде функций, иногда обобщаемых в *шаблонные функции* и *функциональные объекты*.
- В [главе 8](#) дается обзор концептов, методов и языковых особенностей, лежащих в основе обобщённого программирования. Основное внимание уделя-

ется определению и использованию *концептов* для точного определения интерфейсов к шаблонам и направлению разработки. Вводятся *вариативные шаблоны* для определения наиболее общих и гибких интерфейсов.

Здесь описаны языковые средства, поддерживающие идиомы программирования, известные как объектно-ориентированное программирование и обобщённое программирование. В последующих [главах 9-18](#) представлены примеры средств стандартной библиотеки и их использования.

5.1.1 Классы

Центральной языковой особенностью C++ является *класс*. Класс - это пользовательский тип, служащий для представления сущности в коде программы. Всякий раз, когда в нашем проекте программы появляется полезная идея, объект, набор данных и т.д., мы стараемся представить это как класс в программе, чтобы идея существовала в коде, а не только в наших головах, в проектном документе или в некоторых комментариях. Программу, построенную на основе хорошо подобранного набора классов, гораздо легче понять и сделать безошибочной, чем ту, которая строит все непосредственно в терминах встроенных типов. В частности, библиотеки зачастую предлагают именно наборы классов.

По сути, все языковые средства, помимо фундаментальных типов, операторов и ключевых слов, существуют для того, чтобы помочь определить лучшие классы или более удобно их использовать. Под “лучше” я подразумеваю более корректный, более простой в обслуживании, более эффективный, более элегантный, более простой в использовании, более легкий для чтения и более понятный для рассуждений. Большинство методов программирования основаны на разработке и реализации определенных типов классов. Потребности и вкусы программистов сильно различаются. Следовательно, поддержка классов является обширной. Здесь мы рассмотрим базовую поддержку для трех важных типов классов:

- Конкретные классы ([§5.2](#))
- Абстрактные классы ([§5.3](#))
- Классы в иерархиях классов ([§5.5](#))

Поразительное количество полезных классов оказывается одного из этих трех видов. Еще больше классов можно рассматривать как упрощенные варианты из этих трёх видов или реализованные с использованием комбинаций методов, используемых в этих основных вариантах.

5.2 Конкретные типы

Основная идея *конкретных классов* заключается в том, что они ведут себя “точно так же, как встроенные типы”. Например, тип комплексное число и целое число бесконечной точности во многом похожи на встроенные в него `int`, за исключением, конечно, того, что они имеют свою собственную семантику и наборы операций. Аналогично, `vector` и `string` во многом похожи на встроенные массивы, за исключением того, что они более гибкие и лучше управляются ([§10.2](#), [§11.3](#), [§12.2](#)).

Определяющей характеристикой конкретного типа является то, что его представление является частью его определения. Во многих важных случаях, таких как `vector`, это представление является всего лишь одним или несколькими указателями на данные,

хранящиеся в другом месте, но это представление присутствует в каждом объекте конкретного класса. Это позволяет реализации быть оптимально эффективной по быстродействию и памяти. В частности, это позволяет нам

- Размещать объекты конкретных типов в стеке, в статически выделяемой памяти и в других объектах (§1.5).
- Ссылаться на объекты напрямую (а не только через указатели или ссылки).
- Инициализировать объекты немедленно и полностью (например, с помощью конструкторов; §2.3).
- Копировать и перемещать объекты (§6.2).

Представление может быть приватным и доступным только через методы (как это имеет место для **Vector**; §2.3), но оно присутствует. Следовательно, если представление изменяется каким-либо существенным образом, пользователь должен выполнить перекompиляцию. Это цена того, что конкретные типы ведут себя точно так же, как встроенные типы. Для типов, которые меняются нечасто, и где локальные переменные обеспечивают столь необходимую ясность и эффективность, это приемлемо и часто идеально. Чтобы повысить гибкость, конкретный тип может сохранять основные части своего представления в динамической памяти и получать к ним доступ через часть, хранящуюся в самом объекте класса. Именно так реализованы **vector** и **string**; их можно рассматривать как дескрипторы ресурсов с тщательно проработанными интерфейсами.

5.2.1 Арифметические типы

“Классическим пользовательским арифметическим типом” является комплексное число **complex**:

```
class complex {
    double re, im;    // representation: two doubles
public:
    complex(double r, double i) :re{r}, im{i} {}    // construct complex from two scalars
    complex(double r) :re{r}, im{0} {}              // construct complex from one scalar
    complex() :re{0}, im{0} {}                       // default complex: {0,0}
    complex(complex z) :re{z.re}, im{z.im} {}        // copy constructor

    double real() const { return re; }
    void real(double d) { re=d; }
    double imag() const { return im; }
    void imag(double d) { im=d; }

    complex& operator+=(complex z)
    {
        re+=z.re;                // add to re and im
        im+=z.im;
        return *this;            // return the result
    }

    complex& operator-=(complex z)
    {
        re-=z.re;
        im-=z.im;
        return *this;
    }

    complex& operator*=(complex);    // defined out-of-class somewhere
```

```
complex& operator/=(complex);      // defined out-of-class somewhere
};
```

Это упрощенная версия `complex` из стандартной библиотеки (§17.4). Само определение класса содержит только операции, требующие доступа к представлению. Представление простое и общепринятое. По практическим соображениям оно должно быть совместимо с тем, что предоставлял Fortran 60 лет назад, и нам нужен обычный набор операторов. В дополнение к требованиям к базовой логике, `complex` должен быть эффективным, иначе он будет непригодным к использованию. Это подразумевает, что простые операции должны быть встраиваемыми (`inline`). То есть простые операции (такие как конструкторы, `+=` и `imag()`) должны быть реализованы без вызовов функций в сгенерированном машинном коде. Функции, определенные в классе, `inline` по умолчанию. Можно явно запросить встраивание, указав перед объявлением функции ключевое слово `inline`. `complex` из библиотеки Industrial Strength (подобный `complex` стандартной библиотеки) тщательно реализован для выполнения соответствующего встраивания. Кроме того, в `complex` стандартной библиотеки функции, показанные здесь, объявлены `constexpr`, чтобы мы могли выполнять сложную арифметику во время компиляции.

Копирующий оператор присваивания и конструктор копирования определены неявно (§6.2).

Конструктор, который может быть вызван без аргумента, называется *конструктором по умолчанию*. Таким образом, `complex()` является конструктором `complex` по умолчанию. Определяя конструктор по умолчанию, вы исключаете возможность существования неинициализированных переменных этого типа.

Спецификаторы `const` для функций, возвращающих действительную и мнимую части, указывают на то, что эти функции не изменяют объект, для которого они вызываются. Функция-член `const` может быть вызвана как для объектов `const`, так и для неконстантных объектов, но неконстантная функция-член может быть вызвана только для неконстантных объектов. Например:

```
complex z = {1,0};
const complex cz {1,3};
z = cz;           // OK: assigning to a non-const variable
cz = z;           // error: assignment to a const
double x = z.real(); // OK: complex::real() is const
```

Множество полезных операций не требуют прямого доступа к представлению `complex`, поэтому они могут быть определены отдельно от определения класса:

```
complex operator+(complex a, complex b) { return a+=b; }
complex operator-(complex a, complex b) { return a-=b; }
complex operator-(complex a) { return {-a.real(), -a.imag()}; } // unary minus
complex operator*(complex a, complex b) { return a*=b; }
complex operator/(complex a, complex b) { return a/=b; }
```

Здесь я использую тот факт, что аргумент, передаваемый по значению, копируется, так что я могу изменить аргумент, не затрагивая копию вызывающего объекта, и использовать результат в качестве возвращаемого значения.

Определения `==` и `!=` просты:

```
bool operator==(complex a, complex b) { return a.real()==b.real() &&
a.imag()==b.imag(); } // equal
bool operator!=(complex a, complex b) { return !(a==b); } // not equal
```

Класс `complex` может быть использован следующим образом:

```
void f(complex z)
{
```

```

complex a {2.3};           // construct {2.3,0.0} from 2.3
complex b {1/a};
complex c {a+z*complex{1,2.3}};
if (c != b)
    c = -(b/a)+2*b;
}

```

Компилятор преобразует операторы, включающие **complex** комплексные числа, в соответствующие вызовы функций. Например, **c!=b** означает **operator!=(c,b)**, и **1/a** означает **operator/(complex{1},a)**.

Определяемые пользователем операторы (“перегруженные операторы”) следует использовать осторожно и традиционно (§6.4). Синтаксис фиксирован языком, поэтому вы не можете определить унарный **/**. Кроме того, невозможно изменить значение оператора для встроенных типов, поэтому вы не можете переопределить **+** для вычитания целых чисел **int**.

5.2.2 Контейнеры

Контейнер - это объект, содержащий коллекцию элементов. Мы называем класс **Vector** контейнером, потому что объекты типа **Vector** являются контейнерами. Как определено в §2.3, **Vector** не является необоснованным контейнером **double**: он прост для понимания, устанавливает полезный инвариант (§4.3), обеспечивает доступ с проверкой диапазона (§4.2) и предоставляет **size()**, что позволяет нам перебирать его элементы. Однако у него есть фатальный недостаток: он выделяет элементы, используя **new**, но никогда не освобождает их. Это не очень хорошая идея, потому что C++ не предлагает сборщик мусора, чтобы сделать неиспользуемую память доступной для новых объектов. В некоторых средах вы не можете использовать сборщик, и часто вы предпочитаете более точное управление уничтожением по логичным соображениям или соображениям производительности. Нам нужен механизм, гарантирующий освобождение памяти, выделенной конструктором; этот механизм называется *деструктором*:

```

class Vector {
public:
    Vector(int s) :elem{new double[s]}, sz{s}    // constructor: acquire resources
    {
        for (int i=0; i!=s; ++i)                // initialize elements
            elem[i]=0;
    }

    ~Vector() { delete[] elem; }                 // destructor: release resources

    double& operator[](int i);
    int size() const;
private:
    double* elem;                               // elem points to an array of sz doubles
    int sz;
};

```

Именем деструктора является оператор дополнения, **~**, за которым следует имя класса; деструктор — это дополнение конструктора.

Конструктор **Vector** выделяет некоторую память в *динамической памяти* (также называемой *кучей*) с помощью оператора **new**. Деструктор выполняет очистку, освобождая эту память с помощью оператора **delete[]**. Простой **delete** удаляет отдельный объект; **delete[]** удаляет массив.

Все это делается без вмешательства пользователей **Vector**. Пользователи просто создают и используют **Vector** так же, как они использовали бы переменные встроенных типов. Например:

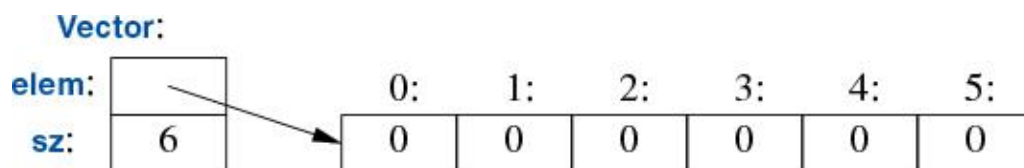
```
Vector gv(10);           // global variable; gv is destroyed at the end of the program
```

```
Vector* gp = new Vector(100);    // Vector on free store; never implicitly destroyed
```

```
void fct(int n)
{
    Vector v(n);
    // ... use v ...
    {
        Vector v2(2*n);
        // ... use v and v2 ...
    } // v2 is destroyed here
    // ... use v ..
} // v is destroyed here
```

Vector подчиняется тем же правилам именования, области видимости, выделения памяти, времени жизни и т.д. (§1.5), как и встроенный тип, такой как **int** и **char**. Этот **Vector** был упрощен за счет исключения обработки ошибок; см. §4.4.

Комбинация конструктора и деструктора лежит в основе многих элегантных методов. В частности, это основа для большинства общих методов управления ресурсами C++ (§6.3, §15.2.1). Рассмотрим графическую иллюстрацию **Vector**:



Конструктор аллоцирует (выделяет память) и соответствующим образом инициализирует элементы **Vector**. Деструктор освобождает память выделенную под элементы. Эта модель обращения к данным очень часто используется для управления данными, размер которых может изменяться в течение срока жизни объекта. Метод получения ресурсов в конструкторе и освобождения их в деструкторе, известный как *получение ресурсов есть инициализация* или *RAII*. Это позволяет нам исключить операции с “голым **new**”, то есть избежать аллокаций в основном коде и сохранить их скрытыми внутри реализации хорошо управляемых абстракций. Аналогичным образом, следует избегать операций с “голым **delete**”. Отказ от голых **new** и **delete** делает код гораздо менее подверженным ошибкам и более защищенным от утечек ресурсов (§15.2.1).

5.2.3 Инициализация контейнеров

Контейнер существует для хранения элементов, поэтому, очевидно, нам нужны удобные способы добавления элементов в контейнер. Мы можем создать **Vector** с соответствующим количеством элементов, а позже присвоить им значения, но обычно другие способы более элегантны. Здесь я просто упомяну два предпочтительных:

- *Конструктор от списка инициализации*: Инициализация с помощью списка элементов.
- **push_back()**: Добавление нового элемента в конец последовательности.

Они могут быть объявлены следующим образом:

```

class Vector {
public:
    Vector();                // default initialize to "empty"; that is, to no ele-
ments
    Vector(std::initializer_list<double>);    // initialize with a list of dou-
bles
    // ...
    void push_back(double);    // add element at end, increasing the size by
one
    // ...
};

```

Функция `push_back()` полезна для добавления произвольного количества элементов. Например:

```

Vector read(istream& is)
{
    Vector v;
    for (double d; is>>d; )    // read floating-point values into d
        v.push_back(d);        // add d to v
    return v;
}

```

Цикл ввода прерывается в случае окончания файла или ошибки форматирования. До тех пор, пока этого не произойдет, каждое считанное число добавляется к `Vector` так, чтобы в конце размер `v` был равен количеству считанных элементов. Я использовал оператор `for`, а не более традиционный оператор `while`, чтобы ограничить область действия `d` циклом.

Возврат потенциально огромного объема данных из `read()` может быть дорогостоящим. Чтобы гарантировать, что возврат `Vector` будет дешевым, необходимо реализовать для него конструктор перемещения (§6.2.2):

```

Vector v = read(cin);    // no copy of Vector elements here

```

Способ реализации `std::vector` для повышения эффективности `push_back()` и других операций, изменяющих размер вектора, представлен в §12.2.

`std::initializer_list`, используемый для определения конструктора от списка инициализации, является типом стандартной библиотеки, известным компилятору: когда мы используем `{}`-список, такой как `{1,2,3,4}`, компилятор создаст объект типа `initializer_list` для передачи программе. Итак, мы можем написать:

```

Vector v1 = {1, 2, 3, 4, 5};    // v1 has 5 elements
Vector v2 = {1.23, 3.45, 6.7, 8};    // v2 has 4 elements

```

Конструктор от списка инициализации `Vector` может быть определен следующим образом:

```

Vector::Vector(std::initializer_list<double> lst)    // initialize with a list
    :elem{new double[lst.size()]}, sz{static_cast<int>(lst.size())}
{
    copy(lst.begin(),lst.end(),elem);    // copy from lst into elem (§13.5)
}

```

К сожалению, стандартная библиотека использует `unsigned` (беззнаковые) целые для размеров и индексов, поэтому нам нужно использовать уродливый `static_cast`, чтобы явно преобразовать размер списка инициализаторов в `int`. Это педантично, потому что вероятность того, что количество элементов в рукописном списке больше, чем наибольшее целое число (32 767 для 16-разрядных целых чисел и 2 147 483 647 для 32-разрядных целых чисел), довольно мала. Однако система типов здравым смыслом не

обладает. Она знает о возможных значениях переменных, а не о фактических значениях, поэтому она может жаловаться там, где фактического нарушения нет. Такие предупреждения иногда могут спасти программиста от серьезной ошибки.

`static_cast` не проверяет значение, которое он преобразует; компилятор считает что программист использует его правильно. Это не всегда верное предположение, поэтому, если вы сомневаетесь, проверьте значение. Явных преобразований типов (часто называемых *casts* (*приведениями*)), чтобы напомнить вам, что они используются в качестве костыля для поддержки чего-то сломанного) лучше избегать. Старайтесь использовать непроверенные приведения только для самого низкого уровня системы. Они подвержены ошибкам.

Другими приведениями являются `reinterpret_cast` и `bit_cast` (§16.7) для обработки объекта как простой последовательности байтов и `const_cast` для “отбрасывания `const`”. Разумное использование системы типов и хорошо продуманных библиотек позволяют нам устранять непроверенные приведения в программном обеспечении более высокого уровня.

5.3 Абстрактные типы

Такие типы, как `complex` и `Vector`, называются *конкретными типами*, потому что их представление является частью их определения. В этом они напоминают встроенные типы. *Абстрактный тип* - напротив, полностью изолирует пользователя от деталей реализации. Для этого, мы отделяем интерфейс от представления и отказываемся от реальных локальных переменных. Поскольку мы ничего не знаем о представлении абстрактного типа (даже о его размере), мы должны размещать объекты в динамической памяти (§5.2.2) и получать к ним доступ через ссылки или указатели (§1.7, §15.2.1).

Сначала мы определяем интерфейс класса `Container`, который мы разработаем как более абстрактную версию нашего `Vector`:

```
class Container {
public:
    virtual double& operator[](int) = 0;    // pure virtual function
    virtual int size() const = 0;           // const member function (§5.2.1)
    virtual ~Container() {}                 // destructor (§5.2.2)
};
```

Этот класс является чистым интерфейсом к конкретным контейнерам, определенным позже. Слово `virtual` означает “может быть переопределено позже в классе, производном от этого”. Неудивительно, что функция, объявленная `virtual`, называется *виртуальной функцией*. Класс, производный от `Container`, предоставляет реализацию интерфейса `Container`. Интересный синтаксис `=0` говорит, что функция является чисто виртуальной; то есть некоторый класс, производный от `Container`, должен определить эту функцию. Таким образом, невозможно определить объект, который является просто `Container`. Например:

```
Container c;                                // error: there can be no objects of an abstract class
Container* p = new Vector_container(10);    // OK: Container is an interface for Vector_container
```

`Container` может служить интерфейсом только для класса, который реализует его функции `operator[]()` и `size()`. Класс как минимум с одной чисто виртуальной функцией называется *абстрактным классом*.

Этот `Container` можно использовать следующим образом:

```

void use(Container& c)
{
    const int sz = c.size();
    for (int i=0; i!=sz; ++i)
        cout << c[i] << '\n';
}

```

Обратите внимание, как `use()` использует интерфейс `Container` при полном незнании деталей реализации. Он использует `size()` и `[]` без какого-либо представления о том, какой именно тип обеспечивает их реализацию. Класс, который предоставляет интерфейс для множества других классов, часто называют *полиморфным типом*.

Как это обычно бывает с абстрактными классами, `Container` не имеет конструктора. В конце концов, у него нет никаких данных для инициализации. С другой стороны, у `Container` действительно есть деструктор, и этот деструктор является виртуальным, так что классы, производные от `Container`, могут предоставлять соответствующие реализации. Опять же, это характерно для абстрактных классов, потому что ими, как правило, манипулируют с помощью ссылок или указателей, и кто-то, уничтожающий `Container` с помощью указателя, понятия не имеет, какими ресурсами владеет его реализация; см. также §5.5.

Абстрактный класс `Container` определяет только интерфейс и никакой реализации. Чтобы контейнер был полезен, мы должны реализовать контейнер, который реализует функции, требуемые его интерфейсом. Для этого мы могли бы использовать конкретный класс `Vector`:

```

class Vector_container : public Container {    // Vector_container implements Container
public:
    Vector_container(int s) : v(s) { }        // Vector of s elements
    ~Vector_container() {}

    double& operator[](int i) override { return v[i]; }
    int size() const override { return v.size(); }
private:
    Vector v;
};

```

Параметр `:public` может быть прочитан как “является производным от” или “является подтипом”. Класс `Vector_container` считается *производным* от класса `Container`, а класс `Container` считается *базовым* для класса `Vector_container`. Альтернативная терминология называет `Vector_container` и `Container` *подклассом* и *суперклассом*, соответственно. Производный класс наследует элементы от своего базового класса, поэтому использование базового и производных классов обычно называют *наследованием*.

Методы `operator[]()` и `size()` *переопределяют* соответствующие элементы в базовом классе `Container`. Я использовал явное указание ключевого слова `override`, чтобы прояснить задуманное. Использование `override` необязательно, но его явное использование позволяет компилятору обнаруживать ошибки, такие как неправильное написание имен функций или незначительные различия между типом `virtual` функции и ее предполагаемым переопределителем. Явное использование `override` особенно полезно в больших иерархиях классов, где в противном случае может быть трудно понять, что должно переопределять что.

Деструктор (`~Vector_container()`) переопределяет деструктор базового класса (`~Container()`). Обратите внимание, что деструктор элемента (`~Vector()`) неявно вызывается деструктором его класса (`~Vector_container()`).

Чтобы такая функция, как `use(Container&)`, использовала `Container` при полном незнании деталей реализации, какая-то другая функция должна будет создать объект, с которым она может работать. Например:

```
void g()
{
    Vector_container vc(10);      // Vector of ten elements
    // ... fill vc ...
    use(vc);
}
```

Поскольку `use()` не знает о `Vector_container`, а знает только интерфейс `Container`, он будет работать так же хорошо для другой реализации `Container`. Например:

```
class List_container : public Container {      // List_container implements Con-
tainer
public:
    List_container() { }                    // empty List
    List_container(initializer_list<double> il) : ld{il} { }
    ~List_container() {}

    double& operator[](int i) override;
    int size() const override { return ld.size(); }
private:
    std::list<double> ld;                    // (standard-library) list of doubles (§12.3)
};

double& List_container::operator[](int i)
{
    for (auto& x : ld) {
        if (i==0)
            return x;
        --i;
    }
    throw out_of_range{"List container"};
}
```

Здесь показан `list<double>` из стандартной библиотеки. Обычно я бы не стал реализовывать контейнер с операцией индекса, используя `list`, потому что производительность операции индекса на `list` ужасна по сравнению с таковой в `vector`. Однако здесь я просто хотел показать реализацию, которая радикально отличается от обычной.

Функция может создать `List_container` и заставить `use()` использовать его:

```
void h()
{
    List_container lc = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    use(lc);
}
```

Дело в том, что `use(Container&)` понятия не имеет, является ли его аргумент `Vector_container`, `List_container` или каким-либо другим типом контейнера; ему не нужно это знать. Для этого можно использовать любой вид `Container`. Он знает только интерфейс, определенный `Container`. Следовательно, `use(Container&)` не нужно перекомпилировать, если изменится реализация `List_container` или используется совершенно новый класс, производный от `Container`.

Оборотной стороной такой гибкости является то, что объектами необходимо управлять с помощью указателей или ссылок (§6.2, §15.2.1).

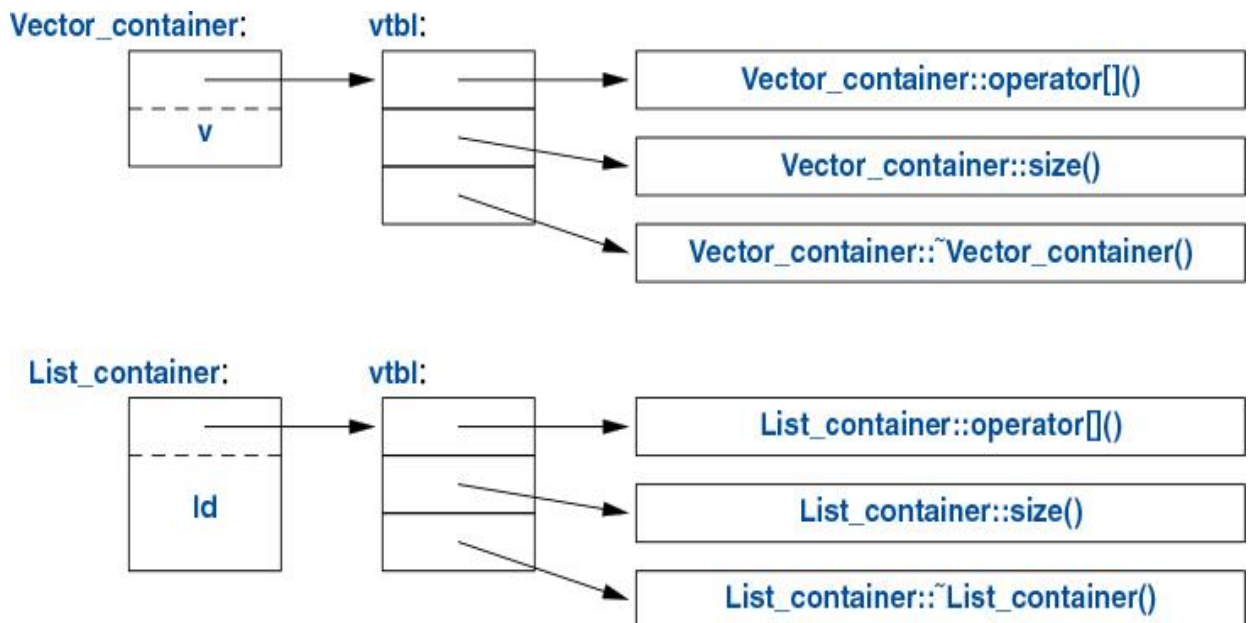
5.4 Виртуальные функции

Опять рассмотрим использование `Container`:

```
void use(Container& c)
{
    const int sz = c.size();

    for (int i=0; i!=sz; ++i)
        cout << c[i] << '\n';
}
```

Как вызов `c[i]` в `use()` преобразуется в соответствующий `operator[]()`? Когда `h()` вызывает `use()`, должен быть вызван `operator[]()` из `List_container`. Когда `g()` вызывает `use()`, должен быть вызван `operator[]()` из `Vector_container`. Чтобы решить эту задачу, объект `Container` должен содержать информацию, позволяющую ему выбрать правильную функцию для вызова во время выполнения. Обычно это реализовано так, что компилятор преобразует имя виртуальной функции в индекс в таблице указателей на функции. Эта таблица обычно называется *таблицей виртуальных функций* или просто *vtbl*. Каждый класс с виртуальными функциями имеет свой собственный *vtbl*, идентифицирующий его виртуальные функции. Это можно представить графически следующим образом:

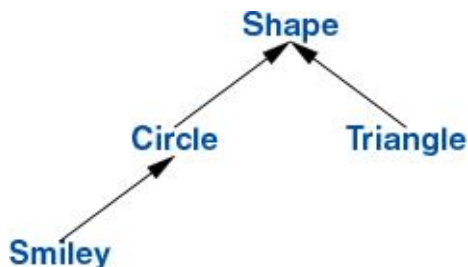


Функции в *vtbl* позволяют корректно использовать объект, даже если размер объекта и расположение его данных неизвестны вызывающей стороне. Реализация вызывающего объекта должна знать только местоположение указателя на *vtbl* в `Container` и индекс, используемый для каждой виртуальной функции. Этот механизм виртуального вызова можно сделать почти таким же эффективным, как механизм “обычного вызова функции” (в пределах 25% и намного дешевле при повторных вызовах одного и того же объекта). Его накладные расходы памяти составляют один указатель на каждый объект класса с виртуальными функциями плюс одна *vtbl* для каждого такого класса.

5.5 Иерархии классов

Пример `Container` - это очень простой пример иерархии классов. *Иерархия классов* - это набор классов, упорядоченных в структуре, созданной путем наследования (например, `: public`). Мы используем иерархии классов для представления понятий, имеющих

иерархические отношения, таких как “Пожарная машина - это разновидность грузовика, который является разновидностью транспортного средства” и “Смайлик - это разновидность круга, который является разновидностью формы”. Огромные иерархии, с сотнями классов, которые являются как глубокими, так и широкими, это обычное явление. В качестве полуреалистичного классического примера давайте рассмотрим геометрические фигуры на экране:



Стрелки представляют отношение наследования. Например, класс **Circle** является производным от класса **Shape**. Иерархия классов обычно рисуется сверху вниз от самого базового класса, корневого, к (определенным позже) производным классам. Чтобы представить эту простую диаграмму в коде, мы должны сначала указать класс, который определяет общие свойства всех фигур:

```

class Shape {
public:
    virtual Point center() const =0;           // pure virtual
    virtual void move(Point to) =0;
    virtual void draw() const = 0;             // draw on current "Canvas"
    virtual void rotate(int angle) = 0;

    virtual ~Shape() {}                        // destructor
    // ...
};
  
```

Естественно, этот интерфейс является абстрактным классом: что касается представления, то ничто (кроме расположения указателя на **vtbl**) не является общим для каждой фигуры **Shape**. Учитывая это определение, мы можем написать общие функции, манипулирующие векторами указателей на фигуры:

```

void rotate_all(vector<Shape*>& v, int angle) // rotate v's elements by angle degrees
{
    for (auto p : v)
        p->rotate(angle);
}
  
```

Чтобы определить конкретную фигуру, мы должны сказать, что это **Shape**, и указать ее конкретные свойства (включая ее виртуальные функции):

```

class Circle : public Shape {
public:
    Circle(Point p, int rad) :x{p}. r{rad} {}           // constructor

    Point center() const override { return x; }
    void move(Point to) override { x = to; }
    void draw() const override;
    void rotate(int) override {}                        // nice simple algorithm
private:
    Point x;      // center
    int r;        // radius
};
  
```


Пока что пример **Shape** и **Circle** не дает ничего нового по сравнению с примером **Container** и **Vector_container**, но мы можем продолжить:

```
class Smiley : public Circle {           // use the circle as the base for a face
public:
    Smiley(Point p, int rad) : Circle{p,rad}, mouth{nullptr} { }
    ~Smiley()
    {
        delete mouth;
        for (auto p : eyes)
            delete p;
    }

    void move(Point to) override;

    void draw() const override;
    void rotate(int) override;

    void add_eye(Shape* s)
    {
        eyes.push_back(s);
    }

    void set_mouth(Shape* s);
    virtual void wink(int i);           // wink eye number i
    // ...
private:
    vector<Shape*> eyes;                 // usually two eyes
    Shape* mouth;
};
```

Метод **push_back()** класса **vector** копирует свой аргумент в **vector** (здесь **eyes**) в качестве последнего элемента, увеличивая размер этого вектора на единицу.

Теперь мы можем определить **Smiley::draw()**, используя вызовы к базовому элементу **Smiley** и методу **draw()**:

```
void Smiley::draw() const
{
    Circle::draw();
    for (auto p : eyes)
        p->draw();
    mouth->draw();
}
```

Обратите внимание на то, как **Smiley** сохраняет свои глаза в **vector** стандартной библиотеки и удаляет их в своем деструкторе. Деструктор **Shape** является **virtual**, и деструктор **Smiley** переопределяет его. Виртуальный деструктор необходим для абстрактного класса, поскольку управление объектом производного класса обычно осуществляется через интерфейс, предоставляемый его абстрактным базовым классом. В частности, он может быть **delete** через указатель на базовый класс. Затем механизм вызова виртуальной функции гарантирует, что будет вызван соответствующий деструктор. Затем этот деструктор неявно вызывает деструкторы своих баз и членов.

В этом упрощенном примере задача программиста - соответствующим образом разместить глаза и рот внутри круга, представляющего лицо.

Мы можем добавлять элементы данных, операции или и то, и другое по мере определения нового класса путем наследования. Это обеспечивает большую гибкость и соответствующие возможности для путаницы и плохого дизайна.

5.5.1 Преимущества иерархий

Иерархия классов дает два вида преимуществ:

- *Наследование интерфейса*: объект производного класса может использоваться везде, где требуется объект базового класса. То есть базовый класс действует как интерфейс для производного класса. Например, классы **Container** и **Shape**. Такие классы часто являются абстрактными классами
- *Наследование реализации*: базовый класс предоставляет функции или данные, которые упрощают реализацию производных классов. Например **Smiley** использует конструктор **Circle** и **Circle::draw()**. Такие базовые классы часто содержат элементы данных и конструкторы.

Конкретные классы – особенно классы с небольшими представлениями – во многом похожи на встроенные типы: мы определяем их как локальные переменные, получаем к ним доступ, используя их имена, копируем их повсюду и т.д. Классы в иерархии классов различаются: мы обычно размещаем их в динамической памяти, используя **new**, и получаем к ним доступ через указатели или ссылки. Например, рассмотрим функцию, которая считывает данные, описывающие фигуры, из входного потока и создает соответствующие объекты **Shape**:

```
enum class Kind { circle, triangle, smiley };

Shape* read_shape(istream& is)    // read shape descriptions from input stream is
{
    // ... read shape header from is and find its Kind k ...
    switch (k) {
    case Kind::circle:
        // ... read circle data {Point,int} into p and r ...
        return new Circle{p,r};
    case Kind::triangle:
        // ... read triangle data {Point,Point,Point} into p1, p2, and p3 ...
        return new Triangle{p1,p2,p3};
    case Kind::smiley:
        // ... read smiley data {Point,int,Shape,Shape,Shape} into p, r, e1, e2, and m
        Smiley* ps = new Smiley{p,r};
        ps->add_eye(e1);
        ps->add_eye(e2);
        ps->set_mouth(m);
        return ps;
    }
}
```

Программа может использовать этот считыватель фигур следующим образом:

```
void user()
{
    std::vector<Shape*> v;

    while (cin)
        v.push_back(read_shape(cin));

    draw_all(v);                // call draw() for each element
    rotate_all(v,45);           // call rotate(45) for each element

    for (auto p : v)            // remember to delete elements
        delete p;
}
```

Очевидно, что пример упрощен – особенно в отношении обработки ошибок, – но он наглядно иллюстрирует, что `user()` не имеет абсолютно никакого представления о том, какими типами фигур он манипулирует. Код `user()` может быть скомпилирован один раз и позже использован для новых фигур `Shape`, добавленных в программу. Обратите внимание, что нет указателей на фигуры вне `user()`, поэтому `user()` несет ответственность за их освобождение. Это делается с помощью оператора `delete` и критически зависит от виртуального деструктора `Shape`. Поскольку этот деструктор является виртуальным, `delete` вызывает деструктор для самого производного класса. Это крайне важно, поскольку производный класс может получить любые виды ресурсов (таких как дескрипторы файлов, блокировки и выходные потоки), которые необходимо освободить. В этом случае `Smiley` удаляет объекты `eyes` и `mouth`. Как только он это сделает, он вызывает деструктор `Circle`. Объекты создаются конструкторами “снизу-вверх” (сначала базовые) и уничтожаются деструкторами “сверху-вниз” (сначала производные).

5.5.2 Навигация в иерархии

Функция `read_shape()` возвращает `Shape*`, так что мы можем обрабатывать все `Shape` одинаково. Однако, что мы можем сделать, если хотим использовать функцию-член, которая предоставляется только определенным производным классом, например, `Smiley wink()`? Мы можем спросить: “Является ли эта `Shape` чем-то вроде `Smiley`?”, используя оператор `dynamic_cast`:

```
Shape* ps {read_shape(cin)};

if (Smiley* p = dynamic_cast<Smiley*>(ps)) { // does ps point to a Smiley?
    // ... a Smiley; use it ...
}
else {
    // ... not a Smiley, try something else ...
}
```

Если во время выполнения объект, на который указывает аргумент `dynamic_cast` (здесь, `ps`), не относится к ожидаемому типу (здесь, `Smiley`) или является классом, производным от ожидаемого, `dynamic_cast` возвращает `nullptr`.

Мы используем `dynamic_cast` для указателя, когда указатель на объект производного класса является допустимым аргументом. Затем мы проверяем, является ли результат `nullptr`. Этот тест часто удобно использовать при инициализации переменной в условии.

Когда другой тип неприемлем, мы можем просто преобразовать `dynamic_cast` в ссылочный тип. Если объект не относится к ожидаемому типу, `dynamic_cast` бросает исключение `bad_cast`:

```
Shape* ps {read_shape(cin)};
Smiley& r {dynamic_cast<Smiley&>(*ps)}; // somewhere, catch std::bad_cast
```

Код становится чище, когда `dynamic_cast` используется ограниченно. Если мы сможем избежать проверки информации о типе во время выполнения, мы сможем написать более простой и эффективный код, но иногда информация о типе теряется и должна быть восстановлена. Обычно это происходит, когда мы передаем объект некоторой системе, которая принимает интерфейс, указанный базовым классом. Когда эта система позже передаст объект обратно нам, нам, возможно, придется восстановить исходный тип. Операции, аналогичные `dynamic_cast`, известны как операции “является производным от” или “является экземпляром”.

5.5.3 Предотвращение утечки ресурсов

Утечка - это общепринятый термин для обозначения того, что происходит, когда мы приобретаем ресурс и не можем его освободить. Необходимо избегать утечки ресурсов, поскольку это делает утекший ресурс недоступным для системы. Таким образом, утечки могут в конечном итоге привести к замедлению работы или даже сбоям, поскольку в системе заканчиваются необходимые ресурсы.

Опытные программисты наверняка заметили, что я оставил открытыми три возможности для ошибок в примере со **Smiley**:

- Разработчик **Smiley** может не удалить **delete** указатель на **mouth**.
- Пользователь **read_shape()** может не **delete** возвращенный указатель.
- Владелец контейнера с указателями **Shape** может не **delete** объекты, на которые ведут указатели.

В этом смысле указатели на объекты, размещенные в динамической памяти, опасны: “обычный старый указатель” не должен использоваться для представления права собственности. Например:

```
void user(int x)
{
    Shape* p = new Circle{Point{0,0},10};
    // ...
    if (x<0) throw Bad_x{};    // potential leak
    if (x==0) return;         // potential leak
    // ...
    delete p;
}
```

Это приведет к утечке, если **x** не будет положительным. Присвоение результата **new** “голому указателю” приводит к возникновению проблем.

Одним из простых решений таких проблем является использование умного указателя **unique_ptr** стандартной библиотеки (§15.2.1) вместо “голого указателя”, когда требуется удаление:

```
class Smiley : public Circle {
    // ...
private:
    vector<unique_ptr<Shape>> eyes; // usually two eyes
    unique_ptr<Shape> mouth;
};
```

Это пример простого, общего и эффективного метода управления ресурсами (§6.3).

В качестве приятного побочного эффекта этого изменения нам больше не нужно определять деструктор для **Smiley**. Компилятор неявно сгенерирует тот, который выполняет требуемое уничтожение **unique_ptr** (§6.3) в **vector**. Код, использующий **unique_ptr**, будет точно таким же эффективным, как и код, правильно использующий сырые указатели.

Теперь рассмотрим пользователей **read_shape()**:

```
unique_ptr<Shape> read_shape(istream& is) // read shape descriptions from input stream
is
{
    // ... read shape header from is and find its Kind k ...

    switch (k) {
    case Kind::circle:
        // ... read circle data {Point,int} into p and r ...
```

```

        return unique_ptr<Shape>{new Circle{p,r}};           // §15.2.1
    // ...
}

void user()
{
    vector<unique_ptr<Shape>> v;
    while (cin)
        v.push_back(read_shape(cin));

    draw_all(v);           // call draw() for each element
    rotate_all(v,45);      // call rotate(45) for each element
} // all Shapes implicitly destroyed

```

Теперь каждый объект принадлежит `unique_ptr`, который `delete` объект, когда он больше не понадобится, то есть когда соответствующий `unique_ptr` выйдет за пределы области видимости.

Чтобы версия `unique_ptr` функции `user()` работала, нам нужны версии `draw_all()` и `rotate_all()`, которые принимают `vector<unique_ptr<Shape>>`. Написание многих таких `_all()` функций может стать утомительным, поэтому в §7.3.2 показана альтернатива.

5.6 Советы

- [1] Выражайте идеи непосредственно в коде; §5.1; [CG: P.1].
- [2] Конкретный тип - это самый простой вид класса. Там, где это применимо, отдавайте предпочтение конкретному типу перед более сложными классами и простыми структурами данных; §5.2; [CG: C.10].
- [3] Используйте конкретные классы для представления простых понятий; §5.2.
- [4] Отдавать предпочтение конкретным классам, а не иерархиям классов для критически важных для производительности компонентов; §5.2.
- [5] Определяйте конструкторы для обработки инициализации объектов; §5.2.1, §6.1.1; [CG: C.40] [CG: C.41].
- [6] Сделайте функцию членом только в том случае, если ей нужен прямой доступ к представлению класса; §5.2.1; [CG: C.4].
- [7] Определите операторы в первую очередь для имитации обычного использования; §5.2.1; [CG: C.160].
- [8] Используйте функции, не являющиеся членами, для симметричных операторов; §5.2.1; [CG: C.161].
- [9] Объявляйте функцию-член, которая не изменяет состояние своего объекта `const`; §5.2.1.
- [10] Если конструктор приобретает ресурс, его классу нужен деструктор для освобождения ресурса; §5.2.2; [CG: C.20].
- [11] Избегайте операций с “голыми” `new` и `delete`; §5.2.2; [CG: R.11].
- [12] Используйте дескрипторы ресурсов и идиому RAII для управления ресурсами; §5.2.2; [CG: R.1].
- [13] Если класс является контейнером, дайте ему конструктор от списка инициализаторов; §5.2.3; [CG: C.103].
- [14] Используйте абстрактные классы в качестве интерфейсов, когда необходимо полное разделение интерфейса и реализации; §5.3; [CG: C.122].
- [15] Обращайтесь к полиморфным объектам с помощью указателей и ссылок; §5.3.
- [16] Абстрактному классу обычно не нужен конструктор; §5.3; [CG: C.126].

- [17] Используйте иерархии классов для представления концепций с присущей им иерархической структурой; §5.5.
- [18] Класс с виртуальной функцией должен иметь виртуальный деструктор; §5.5; [CG: C.127].
- [19] Используйте `override`, чтобы сделать переопределение явным в больших иерархиях классов; §5.3; [CG: C.128].
- [20] При проектировании иерархии классов проводите различие между наследованием реализации и наследованием интерфейса; §5.5.1; [CG: C.129].
- [21] Используйте `dynamic_cast` там, где навигация по иерархии классов неизбежна; §5.5.2; [CG: C.146].
- [22] Используйте `dynamic_cast` для ссылок, когда невозможность найти требуемый класс считается ошибкой; §5.5.2; [CG: C.147].
- [23] Используйте `dynamic_cast` для указателя, когда невозможность найти требуемый класс считается допустимой альтернативой; §5.5.2; [CG: C.148].
- [24] Используйте `unique_ptr` или `shared_ptr`, чтобы не забывать `delete` объекты, созданные с помощью `new`; §5.5.3; [CG: C.149].

Основные операции

Когда кто-то говорит, что он хочет язык программирования, на котором нужно только написать то, что нужно сделать, дайте ему леденец на палочке.
– Alan Perlis

- [Введение](#)

[Основные операции](#); [Преобразования](#); [Инициализаторы](#)

- [Копирование и перемещение](#)

[Копирующие контейнеры](#); [Перемещающие контейнеры](#)

- [Управление ресурсами](#)
- [Перегрузка операторов](#)
- [Стандартные операции](#)

[Сравнение](#); [Операции для контейнеров](#); [Итераторы и «Умные указатели»](#); [Операторы ввода-вывода](#); `swap()`; `hash<>`

- [Пользовательские литералы](#)
- [Советы](#)

6.1 Введение

Некоторые операции, такие как инициализация, присвоение, копирование и перемещение, являются фундаментальными в том смысле, что правила языка программирования делают предположения относительно них. Другие операции, такие как `==` и `<<`, имеют общепринятые значения, которые опасно игнорировать.

6.1.1 Основные операции

Конструкторы, деструкторы и операции копирования и перемещения для типа логически неразделимы. Мы должны определить их как согласованный набор, иначе возникнут проблемы с логикой или производительностью. Если у класса `x` есть деструктор, который выполняет нетривиальную задачу, такую как освобождение динамической памяти или снятие блокировки, классу, вероятно, потребуется полный набор функций:


```

class X {
public:
    X(Sometype);           // “ordinary constructor”: create an object
    X();                   // default constructor
    X(const X&);           // copy constructor
    X(X&&);                // move constructor
    X& operator=(const X&); // copy assignment: clean up target and copy
    X& operator=(X&&);      // move assignment: clean up target and move
    ~X();                  // destructor: clean up
    // ...
};

```

Существует пять ситуаций, в которых объект может быть скопирован или перемещен:

- В качестве источника присваивания
- В качестве инициализатора объекта
- В качестве аргумента функции
- Как возвращаемое функцией значение
- В качестве исключения

При присваивании используется копирующая или перемещающая версия оператора. В принципе, в других случаях используется конструктор копирования или перемещения. Однако вызов конструктора копирования или перемещения часто оптимизируется путем создания объекта, используемого для инициализации, прямо в целевом объекте. Например:

```

X make(Sometype);
X x = make(value);

```

Здесь компилятор обычно создает **X** из **make()** непосредственно в **x**; таким образом, исключая (“устраняя”) копию (copy elision).

В дополнение к инициализации именованных объектов и объектов в динамической памяти конструкторы используются для инициализации временных объектов и реализации явного преобразования типов.

За исключением “обычного конструктора”, эти специальные функции-члены будут генерироваться компилятором по мере необходимости. Если вы хотите четко указать, как создавать реализации по умолчанию, вы можете:

```

class Y {
public:
    Y(Sometype);
    Y(const Y&) = default; // I really do want the default copy constructor
    Y(Y&&) = default;      // and the default move constructor
    // ...
};

```

Если вы явно укажете некоторые значения по умолчанию, другие определения по умолчанию сгенерированы не будут.

Когда у класса есть поле-указатель, обычно рекомендуется четко указывать операции копирования и перемещения. Причина в том, что указатель может указывать на что-то, что классу необходимо **delete**, и в этом случае копирование по умолчанию по элементам было бы неправильным. В качестве альтернативы, это может указывать на что-то, что класс *не должен delete*. В любом случае, читатель кода хотел бы знать. Пример см. в §6.2.1.

Хорошее эмпирическое правило (иногда называемое *правилом нуля*) состоит в том, чтобы либо определять все основные операции, либо ни одной (используя значение по умолчанию для всех). Например:

```
struct Z {
    Vector v;
    string s;
};

Z z1;           // default initialize z1.v and z1.s
Z z2 = z1;      // default copy z1.v and z1.s
```

Здесь компилятор синтезирует для членов конструктор по умолчанию, конструктор копирования, конструктор перемещения и деструктор по мере необходимости, и все это с правильной семантикой.

В дополнение к `=default` у нас есть `=delete`, чтобы указать, что операция не должна быть сгенерирована. Базовый класс в иерархии классов - это классический пример, когда мы не хотим разрешать копирование по элементам. Например:

```
class Shape {
public:
    Shape(const Shape&) =delete;           // no copying
    Shape& operator=(const Shape&) =delete;
    // ...
};

void copy(Shape& s1, const Shape& s2)
{
    s1 = s2;    // error: Shape copy is deleted
}
```

Конструкция `=delete` приводит к тому, что попытка использования `delete` функции приводит к ошибке во время компиляции; `=delete` может использоваться для запрещения любой функции, а не только основных функций-членов.

6.1.2 Преобразования типов

Конструктор, принимающий один аргумент, определяет преобразование из типа принятого аргумента. Например, `complex` (§5.2.1) предоставляет конструктор из `double`:

```
complex z1 = 3.14;    // z1 becomes {3.14,0.0}
complex z2 = z1*2;    // z2 becomes z1*{2.0,0} == {6.28,0.0}
```

Это неявное преобразование иногда является идеальным, но не всегда. Например, `Vector` (§5.2.2) предоставляет конструктор из `int`:

```
Vector v1 = 7;    // OK: v1 has 7 elements
```

Обычно это считается неудачным, и `vector` стандартной библиотеки не допускает такого “преобразования” `int` в `vector`.

Способ избежать этой проблемы - указать, что разрешено только явное “преобразование”; то есть мы можем определить конструктор следующим образом:

```
class Vector {
public:
    explicit Vector(int s);    // no implicit conversion from int to Vector
    // ...
};
```

Это дает нам:

```
Vector v1(7);    // OK: v1 has 7 elements
Vector v2 = 7;   // error: no implicit conversion from int to Vector
```

Когда дело доходит до преобразований, большинство типов похожи на **Vector**, а не на **complex**, поэтому используйте **explicit** для конструкторов, которые принимают один аргумент, если только нет веской причины не делать этого.

6.1.3 Инициализация элементов

При определении элемента данных класса, мы можем предоставить инициализатор, называемый *инициализатором элемента по умолчанию*. Рассмотрим версию **complex** (§5.2.1):

```
class complex {
    double re = 0;
    double im = 0;    // representation: two doubles with default value 0.0
public:
    complex(double r, double i) :re{r}, im{i} {}    // construct complex from two
                                                    // scalars: {r,i}
    complex(double r) :re{r} {}                    // construct complex from one scalar:
{r,0}
    complex() {}                                    // default complex: {0,0}
    // ...
}
```

Значение по умолчанию используется всякий раз, когда конструктор не предоставляет значения. Это упрощает код и помогает нам избежать случайного оставления элемента неинициализированным.

6.2 Копирование и перемещение

По умолчанию объекты могут быть скопированы. Это верно как для объектов пользовательских типов, так и для встроенных типов. По умолчанию под копированием подразумевается поэлементное копирование: копирование каждого элемента. Например, используя **complex** из §5.2.1:

```
void test(complex z1)
{
    complex z2 {z1};    // copy initialization
    complex z3;
    z3 = z2;            // copy assignment
    // ...
}
```

Теперь **z1**, **z2** и **z3** имеют одинаковое значение, потому что и присваивание, и инициализация скопировали оба элемента.

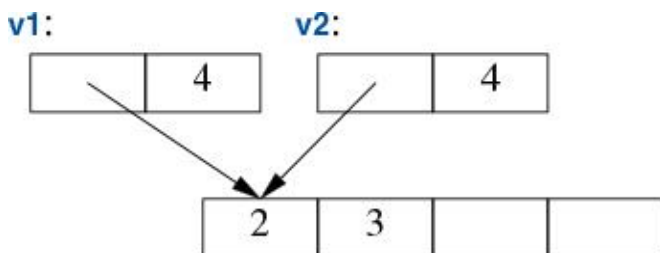
Когда мы разрабатываем класс, мы всегда должны учитывать, может ли объект быть скопирован и как именно. Для простых конкретных типов поэлементное копирование часто является точно подходящей семантикой для копирования. Для некоторых сложных конкретных типов, таких как **Vector**, поэлементное копирование не является подходящей семантикой для копирования; для абстрактных типов поэлементное копирование почти никогда не подходит.

6.2.1 Копирование контейнеров

Когда класс является *дескриптором ресурса*, то есть когда класс отвечает за объект, доступ к которому осуществляется через указатель, поэлементное копирование по умолчанию обычно приводит к сбою. Копирование по элементам нарушило бы инвариант дескриптора ресурса (§4.3). Например, копия по умолчанию оставила бы копию **Vector**, ссылающегося на те же элементы, что и оригинал:

```
void bad_copy(Vector v1)
{
    Vector v2 = v1;           // copy v1's representation into v2
    v1[0] = 2;                // v2[0] is now also 2!
    v2[1] = 3;                // v1[1] is now also 3!
}
```

Предполагая, что **v1** состоит из четырех элементов, результат может быть представлен графически следующим образом:



К счастью, тот факт, что у **Vector** есть деструктор, является сильным намеком на то, что семантика копирования по умолчанию (поэлементная) неверна, и компилятор должен, по крайней мере, предупредить об этом. Нам нужно лучше определить семантику копирования.

Копирование объекта класса определяется двумя членами: *конструктором копирования* и *оператором присваивания копирования*:

```
class Vector {
public:
    Vector(int s);                // constructor: establish invariant, acquire re-
    ~Vector() { delete[] elem; } // destructor: release resources

    Vector(const Vector& a);       // copy constructor
    Vector& operator=(const Vector& a); // copy assignment

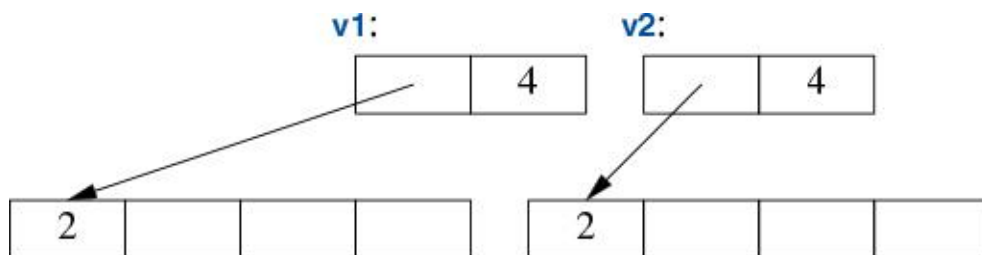
    double& operator[](int i);
    const double& operator[](int i) const;

    int size() const;
private:
    double* elem; // elem points to an array of sz doubles
    int sz;
};
```

Подходящее определение конструктора копирования для **Vector** выделяет пространство для требуемого количества элементов, а затем копирует в него элементы так, чтобы после копирования каждый **Vector** имел свою собственную копию элементов:

```
Vector::Vector(const Vector& a) // copy constructor
    :elem{new double[a.sz]},    // allocate space for elements
    sz{a.sz}
{
    for (int i=0; i!=sz; ++i) // copy elements
        elem[i] = a.elem[i];
}
```

Результат примера `v2=v1` теперь может быть представлен в виде:



Конечно, нам нужен оператор присваивания копированием в дополнение к конструктору копирования:

```
Vector& Vector::operator=(const Vector& a)    // copy assignment
{
    double* p = new double[a.sz];
    for (int i=0; i!=a.sz; ++i)
        p[i] = a.elem[i];
    delete[] elem;          // delete old elements
    elem = p;
    sz = a.sz;
    return *this;
}
```

Имя `this` предопределено в функциях-членах и указывает на объект, для которого вызывается функция-член.

Элементы были скопированы до того, как старые элементы были удалены, так что, если что-то пойдет не так с копией элемента и возникнет исключение, старое значение `Vector` будет сохранено.

6.2.2 Перемещение контейнеров

Мы можем управлять копированием, определив конструктор копирования и оператор присваивания, но копирование может быть дорогостоящим для больших контейнеров. Мы избегаем затрат на копирование, когда передаем объекты в функции с помощью ссылок, но в результате мы не можем вернуть ссылку на локальный объект (локальный объект будет уничтожен к тому времени, когда вызывающий получит возможность взглянуть на него). Рассмотрим:

```
Vector operator+(const Vector& a, const Vector& b)
{
    if (a.size()!=b.size())
        throw Vector_size_mismatch{};

    Vector res(a.size());

    for (int i=0; i!=a.size(); ++i)
        res[i]=a[i]+b[i];
    return res;
}
```

Возврат из `+` включает копирование результата из локальной переменной `res` в какое-либо место, где вызывающий может получить к нему доступ. Мы могли бы использовать этот `+` следующим образом:

```
void f(const Vector& x, const Vector& y, const Vector& z)
{
    Vector r;
    // ...
}
```

```

    r = x+y+z;
    // ...
}

```

Это означало бы копирование **Vector** по крайней мере дважды (по одному для каждого использования оператора **+**). Если **Vector** большой, скажем, 10 000 **double**, это может привести к затруднениям. Самая неприятная часть заключается в том, что **res** в **operator+()** больше никогда не используется после копирования. На самом деле нам не нужна была копия; мы просто хотели получить результат из функции: мы хотели *переместить* **Vector**, а не *копировать* его. К счастью, мы можем заявить об этом намерении:

```

class Vector {
    // ...

    Vector(const Vector& a);           // copy constructor
    Vector& operator=(const Vector& a); // copy assignment

    Vector(Vector&& a);                // move constructor
    Vector& operator=(Vector&& a);     // move assignment
};

```

Учитывая это определение, компилятор выберет *конструктор перемещения* для реализации передачи возвращаемого значения из функции. Это означает, что **r=x+y+z** не будет включать копирование объектов **Vector**. Вместо этого объекты **Vector** просто перемещаются.

Как обычно, конструктор перемещения **Vector** тривиален для определения:

```

Vector::Vector(Vector&& a)
    :elem{a.elem},           // "grab the elements" from a
    sz{a.sz}
{
    a.elem = nullptr;       // now a has no elements
    a.sz = 0;
}

```

&& означает “rvalue ссылка” и является ссылкой, к которой мы можем привязать rvalue. Слово “rvalue” предназначено для дополнения “lvalue”, что примерно означает “что-то, что может отображаться в левой части присваивания” [Страуструп, 2010]. Таким образом, rvalue – это – в первом приближении - значение, которому вы не можете присвоить значение, например, целое число, возвращаемое вызовом функции. Таким образом, rvalue ссылка - это ссылка на что-то, чему никто другой не может присвоить значение, поэтому мы можем безопасно “украсть” его значение. Примером является локальная переменная **res** в **operator+()** для **Vector**.

Конструктор перемещения *не принимает* аргумент **const**: в конце концов, предполагается, что конструктор перемещения удаляет значение из своего аргумента. *Оператор присвоения перемещением* определяется аналогично.

Операция перемещения применяется, когда rvalue ссылка используется в качестве инициализатора или в качестве правой части присваивания.

После перемещения исходный перемещенный объект должен находиться в состоянии, позволяющем запустить деструктор. Как правило, мы также разрешаем присвоение перемещенному объекту. Алгоритмы стандартной библиотеки ([глава 13](#)) предполагают это. Наш **Vector** делает это.

Там, где программист знает, что значение больше не будет использоваться, но компилятор недостаточно умен, чтобы понять это, программист может дополнительно на это указать:

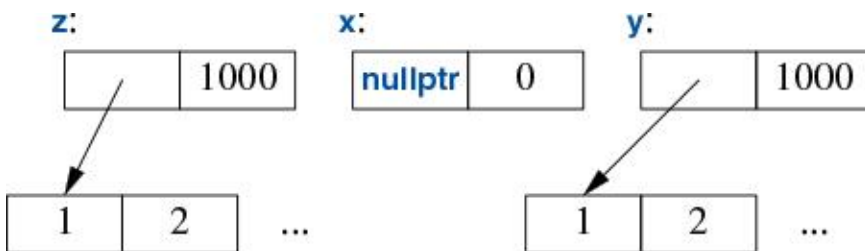
```

Vector f()
{
    Vector x(1000);
    Vector y(2000);
    Vector z(3000);
    z = x;           // we get a copy (x might be used later in f())
    y = std::move(x); // we get a move (move assignment)
    // ... better not use x here ...
    return z;        // we get a move
}

```

Функция стандартной библиотеки `move()` на самом деле ничего не перемещает. Вместо этого он возвращает ссылку на свой аргумент, для которой возможно перемещение – *rvalue* ссылка; это своего рода приведение (§5.2.3).

Непосредственно перед `return` у нас есть:



Когда мы возвращаемся из `f()`, `z` уничтожается после того, как его элементы были перемещены из функции `f()` при `return`. А для `y` деструктор `delete[]` его элементы.

Компилятор обязан (по стандарту C++) оптимизировать большинство копирований, связанных с инициализацией, поэтому конструкторы перемещения вызываются не так часто, как вы могли бы себе представить. Такое *исключение копирования* устраняет даже очень незначительные накладные расходы. С другой стороны, обычно невозможно неявно исключить операции копирования или перемещения из присваиваний, поэтому присваивание перемещением может иметь решающее значение для производительности.

6.3 Управление ресурсами

Определяя конструкторы, операции копирования, перемещения и деструктор, программист может обеспечить полный контроль над временем жизни содержащегося ресурса (например, элементов контейнера). Кроме того, конструктор перемещения позволяет объекту просто и дешево перемещаться из одной области видимости в другую. Таким образом, объекты, которые мы не можем или не хотели бы копировать из области видимости, могут быть просто и дешево перемещены вместо этого. Рассмотрим `thread` стандартной библиотеки, представляющий параллельные вычисления (§18.2) и `Vector` из миллиона `double`. Мы не можем скопировать первое и не хотим копировать второе.

```
std::vector<thread> my_threads;
```

```

Vector init(int n)
{
    thread t {heartbeat};           // run heartbeat concurrently (in a separate
thread)
    my_threads.push_back(std::move(t)); // move t into my_threads (§16.6)
    // ... more initialization ...

    Vector vec(n);
    for (auto& x : vec)

```



```

        x = 777;
    return vec;
}

// move vec out of init()

auto v = init(1 000 000);    // start heartbeat and initialize v

```

Дескрипторы ресурсов, такие как `Vector` и `thread`, во многих случаях являются превосходной альтернативой прямому использованию встроенных указателей. Фактически, “умные указатели” стандартной библиотеки, такие как `unique_ptr`, сами по себе являются дескрипторами ресурсов (§15.2.1).

Я использовал `vector` стандартной библиотеки для хранения `thread`, потому что мы не можем параметризовать наш простой `Vector` с типом элемента до §7.2.

Почти таким же образом, как `new` и `delete` исчезают из кода приложения, мы можем заставить указатели исчезнуть в дескрипторах ресурсов. В обоих случаях результатом является более простой и обслуживаемый код без дополнительных накладных расходов. В частности, мы можем добиться *высокой безопасности ресурсов*; то есть мы можем устранить утечки ресурсов для общего понятия ресурса. Примерами являются `vector`, содержащие память, `thread`, содержащие системные потоки, и `fstream` содержащие дескрипторы файлов.

Во многих языках программирования управление ресурсами в первую очередь делегируется сборщику мусора. В C++ вы можете подключить сборщик мусора. Тем не менее, я считаю сборку мусора последним выбором после того, как были исчерпаны более чистые, общие и лучше локализованные альтернативы управлению ресурсами. Мой идеал - не создавать никакого мусора, тем самым устраняя необходимость в сборщике мусора: Не мусорьте!

Сборка мусора - это, по сути, глобальная схема управления памятью. Продуманные реализации могут компенсировать это, но по мере того, как системы становятся все более распределенными (например, кэши, многоядерные процессоры и кластеры), локальность становится более важной, чем когда-либо.

Кроме того, память - это не единственный ресурс. Ресурс - это все, что должно быть выделено и (явно или неявно) освобождено после использования. Примерами являются память, блокировки, сокеты, дескрипторы файлов и дескрипторы потоков. Неудивительно, что ресурс, который не является просто памятью, называется *ресурсом, не являющимся памятью*. Хорошая система управления ресурсами обрабатывает все виды ресурсов. В любой долго работающей системе необходимо избегать утечек, но чрезмерное удержание ресурсов может быть почти таким же вредным, как и утечка. Например, если система удерживает память, блокировки, файлы и т.д. в течение вдвое большего срока система должна быть обеспечена потенциально вдвое большим количеством ресурсов.

Прежде чем прибегать к сборке мусора, систематически используйте дескрипторы ресурсов: пусть у каждого ресурса есть владелец в некоторой области видимости и по умолчанию он будет освобожден в конце области видимости его владельца. В C++ это известно, как идиома RAII (*получение ресурсов - это инициализация*) и интегрировано с обработкой ошибок в виде исключений. Ресурсы могут быть перемещены в другую область видимости, используя *move-семантику* или “умные указатели”, а совместное владение может быть представлено “общими указателями (*shared pointer*)” (§15.2.1).

В стандартной библиотеке C++ RAII широко распространена: например, для памяти (`string`, `vector`, `map`, `unordered_map` и т.д.), файлов (`ifstream`, `ofstream` и т.д.), потоков (`thread`), блокировок (`lock_guard`, `unique_lock` и т.д.) и общих объектов (через `unique_ptr` и `shared_ptr`). Результатом является неявное управление ресурсами, которое незаметно при обычном использовании и приводит к низкой продолжительности удержания ресурсов.

6.4 Перегрузка операторов

Мы можем придать смысл операторам C++ для пользовательских типов (§2.4, §5.2.1). Это называется *перегрузкой оператора*, потому что при использовании правильная реализация оператора должна быть выбрана из набора операторов с таким же именем. Например, наш `+` для комплексных чисел в `z1+z2` (§5.2.1) следует отличать от `+` для целых и `+` для чисел с плавающей запятой (§1.4.1).

Невозможно определить новые операторы, например, мы не можем определить операторы `^^`, `==`, `**`, `$`, или унарный `%`. Допущение этого вызвало бы столько же путаницы, сколько и пользы.

Настоятельно рекомендуется определять операторы с обычной семантикой. Например, оператор `+`, который вычитает, никому не принесет никакой пользы.

Мы можем определять операторы для пользовательских типов (классов и перечислений):

- Бинарные арифметические операторы: `+`, `-`, `*`, `/`, и `%`
- Бинарные логические операторы: `&` (побитовое И), `|` (побитовое ИЛИ), и `^` (побитовое исключающее ИЛИ)
- Бинарные операторы сравнения: `==`, `!=`, `<`, `<=`, `>`, `>=`, и `<=>`
- Логические операторы: `&&` и `||`
- Унарные арифметические и логические операторы: `+`, `-`, `~` (побитовое НЕ) и `!` (логическое отрицание)
- Присваивание: `=`, `+=`, `*=` и т.д.
- Инкремент и декремент: `++` и `--`
- Операции с указателями: `->`, унарная `*`, и унарный `&`
- Вызов: `()`
- Обращение к элементу: `[]`
- Запятая: `,`
- Сдвиг: `>>` и `<<`

К сожалению, мы не можем переопределить оператор точка (`.`) для получения интеллектуальных ссылок.

Оператор может быть определен как функция-член:

```
class Matrix {  
    // ...  
    Matrix& operator=(const Matrix& a); // assign m to *this; return a reference to  
    *this  
};
```

Обычно это делается для операторов, которые изменяют свой первый операнд, и по историческим причинам требуется для `=`, `->`, `()` и `[]`.

В качестве альтернативы, большинство операторов могут быть определены как автономные функции вне класса:

```
Matrix operator+(const Matrix& m1, const Matrix& m2); // assign m1 to m2 and return  
the sum
```

Принято определять операторы с симметричными операндами как самостоятельные функции, чтобы оба операнда обрабатывались одинаково. Чтобы получить хорошую производительность при возврате потенциально большого объекта, такого как `Matrix`, мы полагаемся на семантику перемещения (§6.2.2).

6.5 Стандартные операции

Некоторые операции имеют общепринятое значение, когда они определены для типа. Эти общепринятые значения часто используются программистами и библиотеками (в частности, стандартной библиотекой), поэтому разумно придерживаться их при разработке новых типов, для которых операции имеют смысл.

- Сравнение: `==`, `!=`, `<`, `<=`, `>`, `>=` и `<=>` (§6.5.1)
- Операторы контейнеров: `size()`, `begin()` и `end()` (§6.5.2)
- Итераторы и “умные указатели”: `->`, `*`, `[]`, `++`, `--`, `+`, `-`, `+=` и `-=` (§13.3, §15.2.1)
- Функции: `()` (§7.3.2)
- Операции ввода-вывода: `>>` и `<<` (§6.5.4)
- `swap()` (§6.5.5)
- Хэш-функции: `hash<>` (§6.5.6)

6.5.1 Операторы сравнения

Имеется ввиду сравнение на равенство (`==` и `!=`) тесно связано с копированием. После копирования копии должны проверяться на равенство:

```
X a = something;
X b = a;
assert(a==b);    // if a!=b here, something is very odd (§4.5)
```

При определении `==` также определите `!=` и убедитесь, что `a!=b` означает `!(a==b)`.

Аналогично, если вы определяете `<`, также определите `<=`, `>`, `>=` чтобы убедиться, что выполняются обычные эквивалентности:

- `a<=b` равнозначно `(a<b)||((a==b) и !(b<a))`.
- `a>b` равнозначно `b<a`.
- `a>=b` равнозначно `(a>b)||((a==b) и !(a<b))`.

Чтобы обеспечить идентичную обработку обоих операндов бинарного оператора, такого как `==`, его лучше всего определить его как отдельную функцию в пространстве имен своего класса. Например:

```
namespace NX {
    class X {
        // ...
    };
    bool operator==(const X&, const X&);
    // ...
};
```

“Оператор космический корабль” `<=>` сам по себе является законом; его правила отличаются от правил для всех других операторов. В частности, при определении значения по умолчанию `<=>` неявно определяются другие операторы сравнения:

```
class R {
    // ...
    auto operator<=>(const R& a) const = default;
};

void user(R r1, R r2)
{
    bool b1 = (r1<=>r2) == 0;    // r1==r2
}
```

```

bool b2 = (r1<=>r2) < 0;    // r1<r2
bool b3 = (r1<=>r2) > 0;    // r1>r2

bool b4 = (r1==r2);
bool b5 = (r1<r2);
}

```

Как и `strcmp()` в C, `<=>` реализует трехстороннее сравнение. Отрицательное возвращаемое значение означает меньше, 0 означает равно, а положительное значение означает больше.

Если `<=>` определено не как по умолчанию, то `==` неявно не определено, но `<` и другие операторы сравнения определены! Например:

```

struct R2 {
    int m;
    auto operator<=>(const R2& a) const { return a.m == m ? 0 : a.m < m ? -1 : 1; }
};

```

Здесь я использовал форму выражения оператора `if: p?x:y` - это выражение, которое вычисляет условие `p`, и если оно истинно, то значение `?` выражения равно `x`, в противном случае `y`.

```

void user(R2 r1, R2 r2)
{
    bool b4 = (r1==r2);    // error: no non-default ==
    bool b5 = (r1<r2);     // OK
}

```

Это приводит к такому шаблону определения для нетривиальных типов:

```

struct R3 { /* ... */ };

auto operator<=>(const R3& a, const R3& b) { /* ... */ }

bool operator==(const R3& a, const R3& b) { /* ... */ }

```

Большинство типов стандартной библиотеки, таких как `string` и `vector`, следуют этому шаблону. Причина в том, что если тип имеет более одного элемента, участвующего в сравнении, по умолчанию `<=>` проверяет их по одному, получая лексикографический порядок. В таком случае часто имеет смысл дополнительно предоставить отдельный оптимизированный `==`, потому что `<=>` должен изучить все элементы, чтобы определить все три альтернативы. Рассмотрим сравнение символьных строк:

```

string s1 = "asdfghjkl";
string s2 = "asdfghjk";

bool b1 = s1==s2;        // false
bool b2 = (s1<=>s2)==0;   // false

```

Используя обычное `==`, мы обнаруживаем, что строки не равны, посмотрев на количество символов. Используя `<=>`, мы должны прочитать все символы `s2`, чтобы обнаружить, что оно меньше `s1` и, следовательно, не равно.

В операторе `<=>` есть еще много деталей, но они в первую очередь представляют интерес для продвинутых разработчиков библиотечных средств, занимающихся сравнениями и сортировкой, выходящими за рамки этой книги. Более старый код не использует `<=>`.

6.5.2 Операции с контейнерами

Если нет действительно веской причины не делать этого, разрабатывайте контейнеры в стиле контейнеров стандартной библиотеки ([глава 12](#)). В частности, сделайте ресурс контейнера безопасным, внедрив его в качестве дескриптора с соответствующими основными операциями (§[6.1.1](#), §[6.2](#)).

Все контейнеры стандартной библиотеки знают свое количество элементов, и мы можем получить его, вызвав `size()`. Например:

```
for (size_t i = 0; i!=c.size(); ++i)    // size_t is the name of the type
    c[i] = 0;                          // returned by a standard-library size()
```

Однако вместо обхода контейнеров с использованием индексов от 0 до `size()` стандартные алгоритмы ([глава 13](#)) полагаются на понятие *последовательностей*, ограниченных парами *итераторов*:

```
for (auto p = c.begin(); p!=c.end(); ++p)
    *p = 0;
```

Здесь `c.begin()` - это итератор, указывающий на первый элемент `c`, а `c.end()` указывает на следующий за последним элемент `c`. Подобно указателям, итераторы поддерживают `++` для перехода к следующему элементу и `*` для доступа к значению элемента, на который указывают.

Функции `begin()` и `end()` также используются реализацией диапазонного `for`, поэтому мы можем упростить циклы по диапазону:

```
for (auto& x : c)
    x = 0;
```

Итераторы используются для передачи последовательностей в алгоритмы стандартной библиотеки. Например:

```
sort(v.begin(),v.end());
```

Эта *итераторная модель* (§[13.3](#)) обеспечивает большую обобщённость и эффективность. Для получения подробной информации и дополнительных операций с контейнерами см. [главы 12](#) и [13](#).

`begin()` и `end()` также могут быть определены как самостоятельные функции; см. §[7.2](#). Версии `begin()` и `end()` для `const` контейнеров называются `cbegin()` и `cend()`.

6.5.3 Итераторы и “умные указатели”

Определяемые пользователем итераторы (§[13.3](#)) и “умные указатели” (§[15.2.1](#)) реализуют операторы и аспекты указателя, необходимые для их целей, и часто добавляют семантику по мере необходимости.

- Доступ: `*`, `->` (для класса), и `[]` (для контейнера)
- Итерация/навигация: `++` (вперёд), `--` (назад), `+=`, `-=`, `+` и `-`
- Копирование и/или перемещение: `=`

6.5.4 Операции ввода-вывода

Для пар целых чисел `<<` означает сдвиг влево, а `>>` означает сдвиг вправо. Однако для `iostream` они являются операторами вывода и ввода данных соответственно (§[1.8](#), [глава 11](#)). Для получения подробной информации и дополнительных операций ввода-вывода см. [главу 11](#).

6.5.5 swap()

Многие алгоритмы, в первую очередь `sort()`, используют функцию `swap()`, которая обменивает значения двух объектов. Такие алгоритмы обычно предполагают, что `swap()` выполняется очень быстро и не выдает исключения. Стандартная библиотека предоставляет `std::swap(a,b)`, реализованный в виде трех операций перемещения (§16.6). Если вы создаете тип, копирование которого обходится дорого и который можно было бы корректно поменять местами (например, с помощью функции сортировки), то назначьте ему операции перемещения или `swap()` или и то, и другое. Обратите внимание, что контейнеры стандартной библиотеки (глава 12) и `string` (§10.2.1) имеют операции быстрого перемещения элементов.

6.5.6 hash<>

`unordered_map<K,V>` из стандартной библиотеки представляет собой хэш-таблицу с `K` в качестве типа ключа и `V` в качестве типа значения (§12.6). Чтобы использовать тип `X` в качестве ключа, мы должны определить `hash<X>`. Для распространенных типов, таких как `std::string`, стандартная библиотека определяет `hash<>` самостоятельно.

6.6 Пользовательские литералы

Одна из целей классов состояла в том, чтобы позволить программисту разрабатывать и реализовывать типы, максимально имитирующие встроенные типы. Конструкторы обеспечивают инициализацию, которая равна или превосходит по гибкости и эффективности инициализацию встроенных типов, а для встроенных типов у нас есть литералы:

- `123` как пример `int`.
- `0xFF00u` как пример `unsigned int`.
- `123.456` в качестве `double`.
- `"Surprise!"` как `const char[10]`.

Может быть полезно предоставить такие литералы и для пользовательского типа. Это делается путем определения значения подходящего к литералу суффикса, так что мы можем получить

- `"Surprise!"s` является `std::string`.
- `123s` это `second`.
- `12.7i` это `imaginary` такое что `12.7i+47` является `complex` числом (например, `{47,12.7}`).

В частности, мы можем получить эти примеры из стандартной библиотеки, используя подходящие заголовки и пространства имен:

Суффиксы стандартной библиотеки для литералов		
<code><chrono></code>	<code>std::literals::chrono_literals</code>	<code>h, min, s, ms, us, ns</code>
<code><string></code>	<code>std::literals::string_literals</code>	<code>s</code>
<code><string_view></code>	<code>std::literals::string_literals</code>	<code>sv</code>
<code><complex></code>	<code>std::literals::complex_literals</code>	<code>i, il, if</code>

Литералы с пользовательскими суффиксами называются *пользовательскими литералами* или *UDL*. Такие литералы определяются с помощью *литеральных операторов*. Ли-

теральный оператор преобразует литерал, переданный как аргумент, за которым следует суффикс, в возвращаемый тип. Например, `i` для `imaginary` суффикса может быть реализован следующим образом:

```
constexpr complex<double> operator""i(long double arg)    // imaginary literal
{
    return {0,arg};
}
```

Где

- `operator""` указывает на то, что мы определяем литеральный оператор.
- `i` после *литерального индикатора* `""` является суффиксом, которому оператор придает значение.
- Тип аргумента `long double` указывает, что суффикс (`i`) определяется для литерала с плавающей запятой.
- Тип возвращаемого значения `complex<double>`, определяет тип результирующего литерала.

Учитывая это, мы можем написать

```
complex<double> z = 2.7182818+6.283185i;
```

Реализация суффикса `i` и `+` оба являются `constexpr`, поэтому вычисление значения `z` выполняется во время компиляции.

6.7 Советы

- [1] Управляйте созданием, копированием, перемещением и уничтожением объектов; §6.1.1; [CG: R.1].
- [2] Спроектируйте конструкторы, присваивания и деструктор как согласованный набор операций; §6.1.1; [CG: C.22].
- [3] Определите либо все основные операции или ни одной; §6.1.1; [CG: C.21].
- [4] Если создаваемые по умолчанию конструктор, присваивания или деструктор подходят, позвольте компилятору сгенерировать их; §6.1.1; [CG: C.20].
- [5] Если у класса есть элемент-указатель, подумайте, нужен ли ему пользовательский или удаленный деструктор, копирование и перемещение; §6.1.1; [CG: C.32] [CG: C.33].
- [6] Если у класса есть определённый пользователем деструктор, ему, вероятно, требуются определенные пользователем или удаленные копирование и перемещение; §6.2.1.
- [7] По умолчанию конструкторы с одним аргументом объявляйте `explicit`; §6.1.2; [CG: C.46].
- [8] Если член класса имеет разумное значение по умолчанию, укажите его в качестве инициализатора элемента данных; §6.1.3; [CG: C.48].
- [9] Переопределите или запретите копирование, если значение по умолчанию не подходит для типа; §6.1.1; [CG: C.61].
- [10] Возвращайте контейнеры по значению (полагаясь на замену копирования перемещением для повышения эффективности); §6.2.2; [CG: F.20].
- [11] Избегайте явного использования `std::copy()`; §16.6; [CG: ES.56].
- [12] Для больших операндов используйте `const` ссылки аргументов; §6.2.2; [CG: F.16].

- [13] Обеспечьте надежную сохранность ресурсов; то есть никогда не допускайте утечки чего-либо, что вы считаете ресурсом; §6.3; [CG: R.1].
- [14] Если класс является дескриптором ресурса, ему нужны определяемые пользователем конструктор, деструктор и операции копирования не по умолчанию; §6.3; [CG: R.1].
- [15] Управляйте всеми ресурсами – памятью и ресурсами, не связанными с памятью, используя RAII; §6.3; [CG: R.1].
- [16] Перегружайте операции для имитации обычного использования; §6.5; [CG: C.160].
- [17] Если вы перегружаете оператор, определите все операции, которые обычно работают вместе; §6.1.1, §6.5.
- [18] Если вы определяете `<=>` для типа как не заданный по умолчанию, также определите `==`; §6.5.1.
- [19] Следуйте дизайну контейнеров стандартной библиотеки; §6.5.2; [CG: C.100].

Шаблоны

*Ваша цитата здесь.
– B. Stroustrup*

- [Введение](#)
- [Параметризованные типы](#)

[Ограниченные аргументы шаблона](#); [Аргументы-значения шаблона](#); [Выведение типов аргументов шаблона](#)

- [Параметризованные операции](#)

[Шаблоны функций](#); [Функциональные объекты](#); [Лямбда-выражения](#)

- [Шаблонные механизмы](#)

[Шаблоны переменных](#); [Псевдонимы](#); [if времени компиляции](#)

- [Советы](#)

7.1 Введение

Тот, кто хочет использовать вектор, вряд ли всегда будет довольствоваться вектором `double`. Вектор - это общее понятие, независимое от понятия числа с плавающей запятой. Следовательно, тип элемента вектора должен быть представлен независимо. *Шаблон* - это класс или функция, которые мы параметризуем с помощью набора типов или значений. Мы используем шаблоны для представления идей, которые лучше всего понимать, как нечто общее, на основе чего мы можем генерировать конкретные типы и функции, указывая аргументы, такие как тип `double` в качестве типа элемента `vector`.

В этой главе основное внимание уделяется языковым механизмам. В [главе 8](#) рассматриваются методы программирования, а в [главах 10-18](#) приводится множество примеров.

7.2 Параметризованные типы

Мы можем обобщить наш тип вектор из `double` (§5.2.2) на тип вектор из любых, сделав его шаблоном `template` и заменив конкретный тип `double` параметром типа. Например:

```
template<typename T>
class Vector {
```

```
private:
    T* elem; // elem points to an array of sz elements of type T
    int sz;
public:
    explicit Vector(int s);          // constructor: establish invariant, acquire re-
sources
    ~Vector() { delete[] elem; }    // destructor: release resources

    // ... copy and move operations ...

    T& operator[](int i);            // for non-const Vectors
    const T& operator[](int i) const; // for const Vectors (§5.2.1)
    int size() const { return sz; }
};
```

Префикс `template<typename T>` делает `T` параметром типа для объявления, которому он предшествует. Это версия C++ математического выражения “для всех `T`” или, точнее, “для всех типов `T`.” Если вам нужен математический “для всех `T`, такой, что `P(T)`”, используйте концепты (§7.2.1, §8.2). Использование `class` для введения параметра типа эквивалентно использованию `typename`, и в более старом коде мы часто видим `template<class T>` в качестве префикса.

Функции-члены могут быть определены аналогичным образом:

```
template<typename T>
Vector<T>::Vector(int s)
{
    if (s<0)
        throw length_error{"Vector constructor: negative size"};
    elem = new T[s];
    sz = s;
}

template<typename T>
const T& Vector<T>::operator[](int i) const
{
    if (i<0 || size()<=i)
        throw out_of_range{"Vector::operator[]"};
    return elem[i];
}
```

Учитывая эти определения, мы можем определить `Vector` следующим образом:

```
Vector<char> vc(200);          // vector of 200 characters
Vector<string> vs(17);         // vector of 17 strings
Vector<list<int>> vli(45);      // vector of 45 lists of integers
```

Символы `>>` в `Vector<list<int>>` завершают вложенные аргументы шаблона (это закрывающие угловые скобки для `<list` и `<int>`); это не ошибочный оператор ввода.

Мы можем использовать `Vector`, таким образом:

```
void write(const Vector<string>& vs)          // Vector of some strings
{
    for (int i = 0; i!=vs.size(); ++i)
        cout << vs[i] << '\n';
}
```

Чтобы обеспечить поддержку диапазонного цикла `for` для нашего `Vector`, мы должны определить подходящие функции `begin()` и `end()`:

```
template<typename T>
T* begin(Vector<T>& x)
```

```

{
    return &x[0];    // pointer to first element or to one-past-the-last element
}

template<typename T>
T* end(Vector<T>& x)
{
    return &x[0]+x.size();    // pointer to one-past-the-last element
}

```

Учитывая это, мы можем написать:

```

void write2(Vector<string>& vs)    // Vector of some strings
{
    for (auto& s : vs)
        cout << s << '\n';
}

```

Аналогично, мы можем определять списки, векторы, карты (то есть ассоциативные массивы), неупорядоченные карты (то есть хэш-таблицы) и т.д. в качестве шаблонов ([глава 12](#)).

Шаблоны - это механизм времени компиляции, поэтому их использование не требует дополнительных затрат времени выполнения по сравнению с кодом, созданным вручную. Фактически, код, сгенерированный для `Vector<double>`, идентичен коду, сгенерированному для версии `Vector` из [главы 5](#). Более того, код, сгенерированный для `vector<double>` стандартной библиотеки, вероятно, будет лучше (поскольку на его реализацию было затрачено больше усилий).

Шаблон плюс набор аргументов шаблона называется *инстанцированием* экземпляра или *специализацией*. В конце процесса компиляции, во время *инстанцирования* экземпляра, генерируется код для каждого экземпляра, используемого в программе (§8.5).

7.2.1 Ограниченные аргументы шаблона

Чаще всего шаблон будет иметь смысл только для аргументов шаблона, которые соответствуют определенным критериям. Например, `Vector` обычно предлагает операцию копирования, и если это так, то он должен требовать, чтобы его элементы поддерживали копирование. То есть мы должны потребовать, чтобы аргументом шаблона `Vector` было не просто `typename`, а `Element`, где “`Element`” определяет требования к типу, который может быть элементом:

```

template<Element T>
class Vector {
private:
    T* elem;    // elem points to an array of sz elements of type T
    int sz;
    // ...
};

```

Этот префикс `template<Element T>` является для C++ математической версией “для всех `T`, таких как `Element(T)`”; то есть `Element` - это предикат, который проверяет, обладает ли `T` всеми свойствами, которые требуются `Vector`. Такой предикат называется *концепт* (§8.2). Аргумент шаблона, для которого задан концепт, называется *ограниченным аргументом*, а шаблон, для которого аргумент ограничен, называется *ограниченным шаблоном*.

Требования к типу элемента стандартной библиотеки немного сложнее (§12.2), но для нашего простого **Vector Element** мог бы быть чем-то вроде концепта **copyable** стандартной библиотеки (§14.5).

Попытка использовать шаблон с типом, который не соответствует его требованиям, является ошибкой времени компиляции. Например:

```
Vector<int> v1;           // OK: we can copy an int
Vector<thread> v2;        // error: we can't copy a standard thread (§18.2)
```

Таким образом, концепты позволяют компилятору выполнять проверку типов в момент использования, выдавая лучшие сообщения об ошибках намного раньше, чем это возможно при использовании неограниченных аргументов шаблона. C++ официально не поддерживал концепты до C++20, поэтому более старый код использует неограниченные аргументы шаблона и позволяет отражать требования лишь в документации. Однако код, сгенерированный из шаблонов, проверяется по типу, так что даже неограниченный шаблонный код так же типобезопасен, как и рукописный код. Для неограниченных параметров эта проверка типа не может быть выполнена до тех пор, пока не будут доступны типы всех задействованных объектов, поэтому проверка может произойти неприятно поздно в процессе компиляции, во время создания экземпляра (§8.5), и сообщения об ошибках часто ужасны.

Проверка концепта - это механизм чисто времени компиляции, и сгенерированный код ничуть не уступает коду из неограниченных шаблонов.

7.2.2 Аргументы-значения шаблона

В дополнение к аргументам-типа шаблон может принимать аргументы-значения. Например:

```
template<typename T, int N>
struct Buffer {
    constexpr int size() { return N; }
    T elem[N];
    // ...
};
```

Аргументы-значения полезны во многих контекстах. Например, **Buffer** позволяет нам создавать буферы произвольного размера без использования динамической памяти:

```
Buffer<char,1024> glob;    // global buffer of characters (statically allocated)

void fct()
{
    Buffer<int,10> buf;      // local buffer of integers (on the stack)
    // ...
}
```

К сожалению, по неясным техническим причинам строковый литерал пока не может быть аргументом-значением шаблона. Однако в некоторых контекстах критически важна возможность параметризации с помощью строковых значений. К счастью, мы можем использовать массив строковых символов (строка в стиле C):

```
template<char* s>
void outs() { cout << s; }

char arr[] = "Weird workaround!";

void use()
```

```

{
    outs<"straightforward use">();           // error (for now)
    outs<arr>();                             // writes: Weird workaround!
}

```

В C++ обычно существует обходной путь; нам не нужна прямая поддержка для каждого варианта использования.

7.2.3 Выведение типов аргументов шаблонов

При определении типа как экземпляра шаблона мы должны указать его аргументы шаблона. Рассмотрим возможное использование шаблона `pair` из стандартной библиотеки:

```
pair<int,double> p = {1, 5.2};
```

Необходимость указывать типы аргументов шаблона может быть утомительной. К счастью, во многих контекстах мы можем просто позволить конструктору `pair` вывести аргументы шаблона из инициализатора:

```
pair p = {1, 5.2};           // p is a pair<int,double>
```

Контейнеры представляют еще одним пример:

```

template<typename T>
class Vector {
public:
    Vector(int);
    Vector(initializer_list<T>);           // initializer-list constructor
    // ...
};

Vector v1 {1, 2, 3};    // deduce v1's element type from the initializer element type:
                        // int
Vector v2 = v1;         // deduce v2's element type from v1's element type: int

auto p = new Vector{1, 2, 3};    // p is a Vector<int>*

Vector<int> v3(1);        // here we need to be explicit about the element type (no ele-
                        // ment
                        // type is mentioned)

```

Очевидно, что это упрощает форму записи и может устранить неприятности, вызванные неправильным вводом избыточных аргументов-типов шаблона. Однако это не панацея. Как и все другие мощные механизмы, выведение (дедукция) может вызывать неожиданности. Рассмотрим:

```

Vector<string> vs {"Hello", "World"};    // OK: Vector<string>
Vector vs1 {"Hello", "World"};           // OK: deduces to Vector<const char*> (Sur-
// prise?)
Vector vs2 {"Hello"s, "World"s};         // OK: deduces to Vector<string>
Vector vs3 {"Hello"s, "World"};          // error: the initializer list is not homoge-
// nous
Vector<string> vs4 {"Hello"s, "World"};    // OK: the element type is explicit

```

Типом строкового литерала в стиле C является `const char*` (§1.7.1). Если это не то, что предназначено для `vs1`, мы должны четко указать тип элемента или использовать суффикс `s`, чтобы сделать его `string` (§10.2).

Если элементы списка инициализаторов имеют разные типы, мы не можем определить уникальный тип элемента, поэтому получаем ошибку неоднозначности.

Иногда нам нужно устранить двусмысленность. Например, `vector` стандартной библиотеки имеет конструктор, который принимает пару итераторов, определяющих последовательность, а также конструктор инициализатора, который может принимать пару значений. Рассмотрим:

```
template<typename T>
class Vector {
public:
    Vector(initializer_list<T>);           // initializer-list constructor

    template<typename Iter>
        Vector(Iter b, Iter e);           // [b:e) iterator-pair constructor

    struct iterator { using value_type = T; /* ... */ };
    iterator begin();

    // ...
};

Vector v1 {1, 2, 3, 4, 5};                // element type is int
Vector v2(v1.begin(),v1.begin()+2);       // a pair of iterators or a pair of values
                                           // (of type iterator)?
Vector v3(9,17);                          // error: ambiguous
```

Мы могли бы решить это с помощью концептов (§8.2), но стандартная библиотека и многие другие важные части кода были написаны за десятилетия до того, как у нас появилась языковая поддержка концептов. Для них нам нужен способ сказать: “пара значений одного и того же типа должна рассматриваться как итераторы”. Добавление *руководства по выведению типа* после объявления `Vector` делает именно это:

```
template<typename Iter>
    Vector(Iter,Iter) -> Vector<typename Iter::value_type>;
```

Теперь мы имеем:

```
Vector v1 {1, 2, 3, 4, 5};                // element type is int
Vector v2(v1.begin(),v1.begin()+2);       // pair-of-iterators: element type is int
Vector v3 {v1.begin(),v1.begin()+2};      // element type is Vector2::iterator
```

Синтаксис инициализатора `{}` всегда отдает предпочтение конструктору `initializer_list` (если он присутствует), поэтому `v3` - это вектор итераторов: `Vector<Vector<int>::iterator>`.

Синтаксис инициализации `()` (§12.2) является обычным для случаев, когда нам не нужен `initializer_list`.

Эффекты руководств по выведению типов часто незаметны, поэтому лучше всего создавать шаблоны классов таким образом, чтобы руководства по дедукции не требовались.

Люди, которым нравятся сокращения, называют “выведение аргумента шаблона класса” *CTAD*.

7.3 Параметризированные операции

Шаблоны имеют гораздо больше применений, чем просто параметризация контейнера с помощью типа элемента. В частности, они широко используются для параметризации как типов, так и алгоритмов в стандартной библиотеке (§12.8, §13.5).

Существует три способа выражения операции, параметризованной типами или значениями:

- Шаблон функции
- Функциональный объект(функтор): объект, который может передавать данные и вызываться как функция
- Лямбда-выражение: сокращенная форма записи функтора

7.3.1 Шаблоны функций

Мы можем написать функцию, которая вычисляет сумму значений элементов любой последовательности, которую может перебрать диапазонный `for` (например, контейнер), следующим образом:

```
template<typename Sequence, typename Value>
Value sum(const Sequence& s, Value v)
{
    for (auto x : s)
        v+=x;
    return v;
}
```

Аргумент шаблона `Value` и аргумент функции `v` предназначены для того, чтобы позволить вызывающей стороне указать тип и начальное значение накопителя (переменной, в которой накапливается сумма):

```
void user(Vector<int>& vi, list<double>& ld, vector<complex<double>>& vc)
{
    int x = sum(vi,0);           // the sum of a vector of ints (add ints)
    double d = sum(vi,0.0);      // the sum of a vector of ints (add doubles)
    double dd = sum(ld,0.0);      // the sum of a list of doubles
    auto z = sum(vc,complex{0.0,0.0}); // the sum of a vector of complex<double>s
}
```

Смысл добавления `int` в `double` состоял бы в том, чтобы изящно обрабатывать сумму, большую, чем самый большой `int`. Обратите внимание, как типы аргументов шаблона для `sum<Sequence, Value>` выводятся из аргументов функции. К счастью, нам не нужно явно указывать эти типы.

Эта функция `sum()` является упрощенной версией `accumulate()` стандартной библиотеки (§17.3).

Шаблон функции может быть функцией-членом, но не `virtual` членом. Компилятор не знал бы всех экземпляров такого шаблона в программе, поэтому он не смог бы сгенерировать `vtbl` (§5.4).

7.3.2 Функциональные объекты

Одним из особенно полезных типов шаблонов является *функциональный объект* (иногда называемый *функтором*), который используется для определения объектов, которые могут вызываться как функции. Например:

```
template<typename T>
class Less_than {
    const T val;           // value to compare against
public:
    Less_than(const T& v) :val{v} { }
```

```
bool operator()(const T& x) const { return x<val; } // call operator
};
```

Функция с именем `operator()` реализует *оператор приложения*, `()`, также называемый “вызов функции” или просто “вызов”.

Мы можем определить именованные переменные типа `Less_than` для некоторого типа аргумента:

```
Less_than lti {42}; // lti(i) will compare i to 42 using < (i<42)
Less_than lts {"Backus"s}; // lts(s) will compare s to "Backus" using <
// (s<"Backus")
Less_than<string> lts2 {"Naur"}; // "Naur" is a C-style string, so we need <string>
to
// get the right <
```

Мы можем вызвать такой объект точно так же, как мы вызываем функцию:

```
void fct(int n, const string& s)
{
    bool b1 = lti(n); // true if n<42
    bool b2 = lts(s); // true if s<"Backus"
    // ...
}
```

Функциональные объекты широко используются в качестве аргументов алгоритмов. Например, мы можем подсчитать количество вхождений значений, для которых предикат возвращает значение `true`:

```
template<typename C, typename P>
int count(const C& c, P pred) // assume that C is a container and P is a
{ // predicate on its elements
    int cnt = 0;
    for (const auto& x : c)
        if (pred(x))
            ++cnt;
    return cnt;
}
```

Это упрощенная версия алгоритма `count_if` стандартной библиотеки (§13.5).

Учитывая концепты (§8.2), мы можем формализовать предположения `count()` относительно его аргумента и проверить их во время компиляции.

Предикат - это то, что мы можем вызвать, чтобы вернуть значение `true` или `false`. Например:

```
void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s)
{
    cout << "number of values less than " << x << ": " << count(vec, Less_than{x}) <<
'\n';
    cout << "number of values less than " << s << ": " << count(lst, Less_than{s}) <<
'\n';
}
```

Здесь `Less_than{x}` создает объект типа `Less_than<int>`, для которого оператор вызова сравнивает значения из вектора с `x`; `Less_than{s}` создает объект, который сравнивает значения из списка с `s`.

Прелесть функциональных объектов в том, что они несут в себе значение, с которым происходит сравнение. Нам не нужно писать отдельную функцию для каждого значения (и каждого типа), и нам не нужно вводить неприятные глобальные переменные для хранения значений. Кроме того, простой функциональный объект, такого как

`Less_than`, легко инлайнится, поэтому вызов `Less_than` намного эффективнее, чем косвенный вызов функции. Способность переносить данные плюс их эффективность делают функциональные объекты особенно полезными в качестве аргументов для алгоритмов.

Функциональные объекты, используемые для указания значения ключевых операций общего алгоритма (такие как `Less_than` для `count()`), иногда называются *объектами политики*.

7.3.3 Лямбда выражения

В §7.3.2 мы определили `Less_than` отдельно от его использования. Это может быть неудобно. Следовательно, существует обозначение для неявно генерируемых объектов функций:

```
void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s)
{
    cout << "number of values less than " << x
          << ": " << count(vec,[&](int a){ return a<x; })
          << '\n';

    cout << "number of values less than " << s
          << ": " << count(lst,[&](const string& a){ return a<s; })
          << '\n';
}
```

Запись `[&](int a){ return a<x; }` называется *лямбда-выражением*. Он генерирует функциональный объект, подобный `Less_than<int>{x}`. `[&]` - это *список захвата*, указывающий, что доступ ко всем локальным именам, используемым в теле лямбды (таким как `x`), будет осуществляться через ссылки. Если бы мы хотели “захватить” только `x`, мы могли бы написать так: `[x]`. Если бы мы хотели предоставить сгенерированному объекту копию `x`, мы могли бы написать так: `[x]`. Не захватывать ничего: `[]`, захватывать все локальные переменные по ссылке: `[&]`, а захватывать все локальные переменные по значению: `[=]`.

Для лямбды, определенной в функции-члене, `[this]` захватывает текущий объект по ссылке, чтобы мы могли ссылаться на члены класса. Если нам нужна копия текущего объекта, мы напишем `[*this]`.

Если мы хотим захватить несколько конкретных объектов, мы можем перечислить их. Примером является использование `[i,this]` при использовании `expect()` (§4.5).

7.3.3.1 Лямбды как аргументы функции

Использование лямбд может быть удобным и кратким, но в то же время непонятным. Для нетривиальных действий (скажем, нечто большее, чем простое выражение) я предпочитаю называть операцию так, чтобы более четко указать ее назначение и сделать ее доступной для использования в нескольких местах программы.

В §5.5.3 мы отметили неприятность, связанную с необходимостью написания множества функций для выполнения операций над элементами `vector` указателей и `unique_ptr`, таких как `draw_all()` и `rotate_all()`. Функциональные объекты (в частности, лямбды) могут помочь, позволяя нам отделить обход контейнера от спецификации того, что должно быть сделано с каждым элементом.

Во-первых, нам нужна функция, которая применяет операцию к каждому объекту, на который указывают элементы контейнера указателей:

```
template<typename C, typename Oper>
void for_each(C& c, Oper op)           // assume that C is a container of pointers
```

```

{
    for (auto& x : c)
        op(x);
}
// (see also §8.2.1)
// pass op() a reference to each element pointed to

```

Это упрощенная версия алгоритма стандартной библиотеки `for_each` (§13.5).

Теперь мы можем написать версию `user()` из §5.5 без написания набора функций `_all`:

```

void user()
{
    vector<unique_ptr<Shape>> v;
    while (cin)
        v.push_back(read_shape(cin));
    for_each(v, [](unique_ptr<Shape>& ps){ ps->draw(); }); // draw_all()

    for_each(v, [](unique_ptr<Shape>& ps){ ps->rotate(45); }); // rotate_all(45)
}

```

Я передаю `unique_ptr<Shape>` лямбдам по ссылке. Таким образом, `for_each()` не придется иметь дело с проблемами срока жизни объекта.

Как и функция, лямбда-выражение может быть универсальным. Например:

```

template<class S>
void rotate_and_draw(vector<S>& v, int r)
{
    for_each(v, [](auto& s){ s->rotate(r); s->draw(); });
}

```

Здесь, как и в объявлениях переменных, `auto` означает, что значение любого типа принимается в качестве инициализатора (считается, что аргумент инициализирует формальный параметр в вызове). Это превращает лямбду с параметром `auto` в шаблон, *универсальную лямбду*. При необходимости мы можем ограничить параметр концептом (§8.2). Например, мы могли бы определить `Pointer_to_class` для требования `*` и `->` и записать:

```

for_each(v, [](Pointer_to_class auto& s){ s->rotate(r); s->draw(); });

```

Мы можем вызвать эту универсальную функцию `rotate_and_draw()` с любым контейнером объектов, которые вы можете `draw()` и `rotate()`. Например:

```

void user()
{
    vector<unique_ptr<Shape>> v1;
    vector<Shape*> v2;
    // ...
    rotate_and_draw(v1, 45);
    rotate_and_draw(v2, 90);
}

```

Для еще более тщательной проверки мы могли бы определить концепт `Pointer_to_Shape`, определяющий свойства, которые мы хотим, чтобы тип можно было использовать в качестве формы. Это позволило бы нам использовать фигуры, которые не были производными от класса `Shape`.

7.3.3.2 Лямбды для инициализации

Используя лямбду, мы можем превратить любое утверждение в выражение. В основном это используется для обеспечения операций вычисления значения в качестве значения аргумента, но эта возможность является общей. Рассмотрим сложную инициализацию:

```
enum class Init_mode { zero, seq, cpy, patrn };           // initializer alternatives

void user(Init_mode m, int n, vector<int>& arg, Iterator p, Iterator q)
{
    vector<int> v;

    // messy initialization code:

    switch (m) {
    case zero:
        v = vector<int>(n);    // n elements initialized to 0
        break;
    case cpy:
        v = arg;
        break;
    };

    // ...

    if (m == seq)
        v.assign(p,q);        // copy from sequence [p:q)

    // ...
}
```

Это стилизованный пример, но, к сожалению, не редкий. Нам нужно выбрать один из вариантов для инициализации структуры данных (здесь `v`), и нам нужно выполнить разные вычисления для разных вариантов. Такой код часто является беспорядочным, считается необходимым “для повышения эффективности” и источником ошибок:

- Переменная может быть использована до того, как она получит свое предполагаемое значение.
- “Код инициализации” может быть смешан с другим кодом, что затрудняет его понимание.
- Когда “код инициализации” смешан с другим кодом, легче забыть `case`.
- Это не инициализация, это присвоение (§1.9.2).

Вместо этого мы могли бы преобразовать его в лямбду, используемую в качестве инициализатора:

```
void user(Init_mode m, int n, vector<int>& arg, Iterator p, Iterator q)
{
    vector<int> v = [&] {
        switch (m) {
        case zero:    return vector<int>(n);        // n elements initialized to 0
        case seq:     return vector<int>{p,q};      // copy from sequence [p:q)
        case cpy:     return arg;
        }
    }();
    // ...
}
```

Я все еще “забыл” один `case`, но теперь это легче обнаружить. Во многих случаях компилятор обнаружит проблему и предупредит.

7.3.3.3 Напоследок, `finally()`

Деструкторы предлагают общий и неявный механизм очистки после использования объекта (RAII; §6.3), но что, если нам нужно выполнить некоторую очистку, которая не связана с отдельным объектом, или связана с объектом, у которого нет деструктора (например, потому что это тип, совместно используемый с программой на языке Си)? Мы можем определить функцию `finally()`, которая выполняет действие, подлежащее выполнению при выходе из области видимости

```
void old_style(int n)
{
    void* p = malloc(n*sizeof(int));           // C-style
    auto act = finally([&]{free(p);});         // call the lambda upon scope exit
    // ...
} // p is implicitly freed upon scope exit
```

Это узкоспециализированно, но намного лучше, чем пытаться правильно и последовательно вызывать `free(p)` при любом выходе из функции.

Функция `finally()` тривиальна:

```
template <class F>
[[nodiscard]] auto finally(F f)
{
    return Final_action{f};
}
```

Я использовал атрибут `[[nodiscard]]`, чтобы гарантировать, что пользователи не забудут скопировать сгенерированное `Final_action` в область, для которой предназначено его действие.

Класс `Final_action`, который предоставляет необходимый деструктор, может выглядеть следующим образом:

```
template <class F>
struct Final_action {
    explicit Final_action(F f) :act(f) {}
    ~Final_action() { act(); }
    F act;
};
```

В библиотеке поддержки Core Guidelines (GSL) есть функция `finally()` и предложение по более сложному механизму `scope_exit` для стандартной библиотеки.

7.4 Механизмы шаблонов

Чтобы определить хорошие шаблоны, нам нужны некоторые вспомогательные языковые средства:

- Значения, зависящие от типа: *шаблоны переменных* (§7.4.1).
- Псевдонимы для типов и шаблонов: *шаблоны псевдонимов* (§7.4.2).
- Механизм выбора времени компиляции: *if constexpr* (§7.4.3).
- Механизм времени компиляции для запроса свойств типов и выражений: *requires-выражения* (§8.2.3).

Кроме того, функции `constexpr` (§1.6) и `static_asserts` (§4.5.2) часто принимают участие в разработке и использовании шаблонов.

Эти базовые механизмы в первую очередь являются инструментами для построения общих, основополагающих абстракций.

7.4.1 Шаблоны переменных

Когда мы используем тип, нам часто нужны константы и значения этого типа. Это, конечно, также имеет место, когда мы используем шаблон класса: когда мы определяем `C<T>`, нам часто нужны константы и переменные типа `C<T>` и других типов, зависящих от `T`. Вот пример из моделирования динамики жидкости [Garcia,2015]:

```
template <class T>
    constexpr T viscosity = 0.4;

template <class T>
    constexpr space_vector<T> external_acceleration = { T{}, T{-9.8}, T{} };

auto vis2 = 2*viscosity<double>;
auto acc = external_acceleration<float>;
```

Здесь `space_vector` - это трехмерный вектор.

Достаточно любопытно, что большинство шаблонов переменных кажутся константами. Но с другой стороны, существует и множество переменных. Терминология не соответствует нашим представлениям о неизменности.

Естественно, мы можем использовать произвольные выражения подходящих типов в качестве инициализаторов. Рассмотрим:

```
template<typename T, typename T2>
constexpr bool Assignable = is_assignable<T&,T2>::value;    // is_assignable is a type trait (§16.4.1)

template<typename T>
void testing()
{
    static_assert(Assignable<T&,double>, "can't assign a double to a T");
    static_assert(Assignable<T&,string>, "can't assign a string to a T");
}
```

После некоторых существенных изменений эта идея стала основой определений концептов (§8.2).

Стандартная библиотека использует шаблоны переменных для предоставления математических констант, таких как `pi` и `log2e` (§17.9).

7.4.2 Псевдонимы

Удивительно часто бывает полезно ввести синоним для типа или шаблона. Например, стандартный заголовок `<cstdint>` содержит определение псевдонима `size_t`, возможно такой:

```
using size_t = unsigned int;
```

Фактический тип с именем `size_t` зависит от реализации, поэтому в другой реализации `size_t` может быть `unsigned long`. Наличие псевдонима `size_t` позволяет программисту писать переносимый код.

Очень часто параметризованный тип предоставляет псевдоним для типов, связанных с их аргументами шаблона. Например:

```
template<typename T>
class Vector {
public:
    using value_type = T;
```



```
// ...
};
```

Фактически, каждый контейнер стандартной библиотеки предоставляет `value_type` в качестве имени типа своих элементов ([глава 12](#)). Это позволяет нам написать код, который будет работать для любого контейнера, соответствующего этому соглашению. Например:

```
template<typename C>
using Value_type = C::value_type;           // the type of C's elements

template<typename Container>
void algo(Container& c)
{
    Vector<Value_type<Container>> vec;       // keep results here
    // ...
}
```

Этот `Value_type` является упрощенной версией `range_value_t` из стандартной библиотеки (§16.4.4). Механизм псевдонимов может быть использован для определения нового шаблона путем привязки некоторых или всех аргументов шаблона. Например:

```
template<typename Key, typename Value>
class Map {
    // ...
};

template<typename Value>
using String_map = Map<string, Value>;

String_map<int> m;           // m is a Map<string, int>
```

7.4.3 if времени компиляции

Рассмотрим возможность написания операции, которая может быть реализована с использованием одной из двух функций `slow_and_safe(T)` или `simple_and_fast(T)`. Таких проблем предостаточно в базовом коде, где важны обобщённость и оптимальная производительность. Если задействована иерархия классов, базовый класс может обеспечить общую операцию `slow_and_safe`, а производный класс может переопределить ее с помощью реализации `simple_and_fast`.

В качестве альтернативы, мы можем использовать `if` времени компиляции:

```
template<typename T>
void update(T& target)
{
    // ...
    if constexpr(is_trivially_copyable_v<T>)
        simple_and_fast(target);           // for "plain old data"
    else
        slow_and_safe(target);              // for more complex types
    // ...
}
```

`is_trivially_copyable_v<T>` - это предикат типа (§16.4.1), который сообщает нам, может ли тип быть тривиально скопирован.

Компилятор проходит только выбранную ветвь `if constexpr`. Это решение обеспечивает оптимальную производительность и локальность оптимизации.

Важно отметить, что `if constexpr` не является механизмом манипулирования текстом и не может использоваться для нарушения обычных правил грамматики, типа и области видимости. Например, вот наивная и неудачная попытка условно обернуть вызов в блок `try`:

```
template<typename T>
void bad(T arg)
{
    if constexpr(!is_trivially_copyable_v<T>)
        try {                                     // Oops, the if extends beyond this line
            g(arg);
        }
    if constexpr(!is_trivially_copyable_v<T>)
        } catch(...) { /* ... */ }               // syntax error
}
```

Разрешение таких манипуляций с текстом может серьезно ухудшить читаемость кода и создать проблемы для инструментов, использующих современные методы представления программ (такие как “абстрактные синтаксические деревья”).

Множество подобных попыток «хакнуть плюсы» не нужны, поскольку доступны более чистые решения, которые не нарушают правила области видимости. Например:

```
template<typename T>
void good(T arg)
{
    if constexpr (is_trivially_copyable_v<T>)
        g(arg);
    else
        try {
            g(arg);
        }
        catch (...) { /* ... */ }
}
```

7.5 Советы

- [1] Используйте шаблоны для выражения алгоритмов, применимых ко многим типам аргументов; §7.1; [CG: T.2].
- [2] Используйте шаблоны для выражения контейнеров; §7.2; [CG: T.3].
- [3] Используйте шаблоны для повышения уровня абстракции кода; §7.2; [CG: T.1].
- [4] Шаблоны типобезопасны, но для неограниченных шаблонов проверка происходит слишком поздно; §7.2.
- [5] Пусть конструкторы или шаблоны функций выводят типы аргументов шаблона класса; §7.2.3.
- [6] Используйте функциональные объекты в качестве аргументов алгоритмов; §7.3.2; [CG: T.40].
- [7] Используйте лямбды, если вам нужен простой функциональный объект только в одном месте; §7.3.2.
- [8] Виртуальная функция не может быть шаблонной функцией-членом; §7.3.1.
- [9] Используйте `finally()` для предоставления RAII для типов без деструкторов, которые требуют “операций очистки”; §7.3.3.3.
- [10] Используйте шаблонные псевдонимы для упрощения обозначения и сокрытия деталей реализации; §7.4.2.
- [11] Используйте `if constexpr` для предоставления альтернативных реализаций без накладных расходов времени выполнения; §7.4.3.

Концепты и обобщенное программирование

*Программирование вы должны
начать с интересных алгорит-
мов.*

– Alexandr Stepanov

- [Введение](#)
- [Концепты](#)

[Использование концептов](#); [Перегрузка основанная на концептах](#); [Допустимый код](#);
[Определение концептов](#); [Концепты и `auto`](#); [Концепты и типы](#)

- [Обобщенное программирование](#)

[Использование концептов](#); [Абстракции с использованием шаблонов](#)

- [Шаблоны с переменным количеством аргументов \(вариадик\)](#)

[Выражения свертки](#); [Передача аргументов](#)

- [Модель компиляции шаблонов](#)
- [Советы](#)

8.1 Введение

Для чего нужны шаблоны? Другими словами, какие методы программирования становятся эффективнее с помощью шаблонов? Шаблоны предлагают:

- Возможность передавать типы (а также значения и шаблоны) в качестве аргументов без потери информации. Это подразумевает большую гибкость в применении, и отличные возможности для встраивания (inline), которые широко используются в текущих реализациях.
- Возможности объединить информацию из разных контекстов во время создания экземпляра (инстанцирования). Что подразумевает возможности оптимизации.
- Возможность передавать значения в качестве аргументов шаблона. Что подразумевает возможности для вычислений во время компиляции.

Другими словами, шаблоны предоставляют мощный механизм для вычислений во время компиляции и манипуляций типами, что может привести к созданию очень компактного и эффективного кода. Помните, что типы (классы) могут содержать как код (§7.3.2), так и значения (§7.2.2).

Первое и наиболее распространенное использование шаблонов - это поддержка *обобщенного программирования*, то есть программирования, ориентированного на разработку, реализацию и использование общих алгоритмов. Здесь “общий” означает, что алгоритм может быть разработан таким образом, чтобы принимать самые разнообразные типы, при условии, что они удовлетворяют требованиям алгоритма к его аргументам. Вместе с концептами шаблоны являются основой поддержки обобщенного программирования в C++. Шаблоны обеспечивают параметрический полиморфизм (времени компиляции).

8.2 Концепты

Рассмотрим функцию `sum()` из §7.3.1:

```
template<typename Seq, typename Value>
Value sum(Seq s, Value v)
{
    for (const auto& x : s)
        v+=x;
    return v;
}
```

Функция `sum()` подразумевает что

- его первым аргументом шаблона является некоторая последовательность элементов, и
- его вторым аргументом шаблона является какое-то число.

Чтобы быть более конкретным, `sum()` может быть вызван для пары аргументов:

- *Последовательность Seq*, которая поддерживает `begin()` и `end()`, для работы диапазонного `for` (§1.7; §14.1).
- *Арифметический тип Value*, который поддерживает `+=`, чтобы можно было суммировать элементы последовательности.

Мы называем такие требования *концептами*.

Типы, удовлетворяющие этому упрощенному требованию (и многим другим) и считающиеся последовательностью (также называемой диапазоном), это, например, `vector`, `list` и `map` стандартной библиотеки. Типы, удовлетворяющие этому упрощенному требованию (и многим другим) и считающиеся арифметическим типом, включают `int`, `double` и `Matrix` (для любого разумного определения `Matrix`). Мы могли бы сказать, что алгоритм `sum()` является обобщенным в двух измерениях: типе структуры данных, используемой для хранения элементов (“последовательность”), и типе элементов.

8.2.1 Использование концептов

Большинство аргументов шаблонов должны соответствовать определенным требованиям для правильной компиляции шаблона и правильной работы сгенерированного кода. То есть большинство шаблонов должны быть шаблонами с ограничениями (§7.2.1). Ключевое слово `typename` является наименее ограничивающим, требующим только, чтобы аргумент был типом. Обычно мы можем сделать лучше. Рассмотрим функцию `sum()` еще раз:

```
template<Sequence Seq, Number Num>
Num sum(Seq s, Num v)
{
    for (const auto& x : s)
```

```

        v+=x;
    return v;
}

```

Так гораздо понятнее. Как только мы определим, что означают понятия `Sequence` и `Number`, компилятор сможет отклонять некорректные вызовы, просматривая только интерфейс `sum()`, а не его реализацию. Это улучшает представление отчетов об ошибках.

Однако спецификация интерфейса `sum()` не является полной: я “забыл” сказать, что мы должны иметь возможность добавлять элементы `Sequence` к `Number`. Мы можем это сделать:

```

template<Sequence Seq, Number Num>
    requires Arithmetic<range_value_t<Seq>,Num>
Num sum(Seq s, Num n);

```

`range_value_t` (§16.4.4) последовательности - это тип элементов в этой последовательности; он взят из стандартной библиотеки, где он называет тип элементов диапазона `range` (§14.1). `Arithmetic<X,Y>` - это концепт, указывающий, что мы можем выполнять арифметику с числами типов `X` и `Y`. Это избавляет нас от случайной попытки вычислить `sum()` для `vector<string>` или `vector<int*>`, все еще принимая `vector<int>` и `vector<complex<double>>`. Как правило, когда алгоритму требуются аргументы разных типов, между этими типами существует взаимосвязь, которую желательно сделать явной.

В этом примере нам понадобился только оператор `+=`, но для простоты и гибкости мы не должны слишком сильно ограничивать наш аргумент шаблона. В частности, возможно, когда-нибудь мы захотим выразить `sum()` в терминах `+` и `=`, а не `+=`, и тогда мы будем счастливы, что использовали общий концепт (здесь, `Arithmetic`), а не узкое требование “наличие `+=`”.

Частичные спецификации, как и в первом `sum()` использующем концепты, могут быть очень полезны. Если спецификация не полная, некоторые ошибки не будут обнаружены до момента создания экземпляра. Однако даже частичные спецификации выражают намерение и необходимы для плавной поэтапной разработки там, где мы изначально не осознаем всех необходимых нам требований. При наличии зрелых библиотек концептов первоначальные спецификации будут близки к идеальным.

Неудивительно, что `requires Arithmetic<range_value_t<Seq>,Num>` называется пунктом требований `requirements`. Обозначение `template<Sequence Seq>` - это просто сокращение для явного использования `requires Sequence<Seq>`. Если бы мне нравилось многословие, я мог бы с таким же успехом написать

```

template<typename Seq, typename Num>
    requires Sequence<Seq> && Number<Num> && Arithmetic<range_value_t<Seq>,Num>
Num sum(Seq s, Num n);

```

С другой стороны, мы могли бы также использовать эквивалентность между двумя обозначениями для записи:

```

template<Sequence Seq, Arithmetic<range_value_t<Seq>> Num>
Num sum(Seq s, Num n);

```

В кодовых базах, где мы еще не можем использовать `concept`, нам приходится обходиться соглашениями об именовании и комментариями, такими как:

```

template<typename Sequence, typename Number>
    // requires Arithmetic<range_value_t<Sequence>,Number>
Number sum(Sequence s, Number n);

```

Какую бы форму записи мы ни выбрали, важно разработать шаблон с семантически значимыми ограничениями на его аргументы (§8.2.4).

8.2.2 Перегрузка основанная на концептах

Как только мы правильно определим шаблоны с их интерфейсами, мы сможем выполнять перегрузку на основе их свойств, так же, как мы это делаем для функций. Рассмотрим слегка упрощенную функцию стандартной библиотеки `advance()`, которая дополняет итератор (§13.3):

```
template<forward_iterator Iter>
void advance(Iter p, int n)                // move p n elements forward
{
    while (n-- > 0)
        ++p;                             // a forward iterator has ++, but not + or +=
}

template<random_access_iterator Iter>
void advance(Iter p, int n)                // move p n elements forward
{
    p += n;                               // a random-access iterator has +=
}
```

Компилятор выберет шаблон с наиболее строгими требованиями, которым удовлетворяют аргументы. В этом случае `list` предоставляет только прямые итераторы, но `vector` предлагает итераторы с произвольным доступом, поэтому мы получаем:

```
void user(vector<int>::iterator vip, list<string>::iterator lsp)
{
    advance(vip, 10);                      // uses the fast advance()
    advance(lsp, 10);                      // uses the slow advance()
}
```

Как и другие перегрузки, это механизм времени компиляции, не предполагающий затрат времени выполнения, и там, где компилятор не находит наилучшего выбора, он выдает ошибку двусмысленности. Правила для перегрузки, основанной на концептах, намного проще, чем правила для общей перегрузки (§1.3). Рассмотрим сначала один аргумент для нескольких альтернативных функций:

- Если аргумент не соответствует концепту, этот вариант не может быть выбран.
- Если аргумент соответствует концепту только для одного варианта, выбирается этот вариант.
- Если аргументы из двух вариантов соответствуют концепту, и один из вариантов более строгий, чем другой (соответствует всем требованиям другого и более), выбирается этот вариант.
- Если аргументы из двух вариантов одинаково хорошо соответствуют концепту, это двусмысленность.

Чтобы был выбран вариант, он должен

- соответствовать требованиям ко всем его аргументам и
- по крайней мере, одинаково хорошо согласоваться со всеми аргументами, как и другие варианты, и
- лучше соответствовать по крайней мере одному аргументу.

8.2.3 Правильный код

Вопрос о том, предлагает ли набор аргументов шаблона то, что шаблон требует от своих параметров шаблона, в конечном счете сводится к тому, являются ли некоторые выражения допустимыми.

Используя выражения `requires`, мы можем проверить, является ли набор выражений допустимым. Например, мы могли бы попытаться написать `advance()` без использования концептов стандартной библиотеки `random_access_iterator`:

```
template<forward_iterator Iter>
    requires requires(Iter p, int i) { p[i]; p+i; }           // Iter has subscripting
and
                                                                // integer addition
void advance(Iter p, int n)      // move p n elements forward
{
    p+=n;
}
```

Нет, `requires requires` - это не опечатка. Первое `requires` означает начало пункта `requirements`, а второе `requires` означает начало выражения `requires`

```
requires(Iter p, int i) { p[i]; p+i; }
```

`requires`-выражение - это предикат, который является `true`, если выражение в нем является допустимым кодом, и `false` - если нет.

Я считаю `requires`-выражения ассемблерным кодом обобщённого программирования. Как и обычный ассемблерный код, `requires`-выражения чрезвычайно гибки и не налагают никакой дисциплины программирования. В той или иной форме они находятся в низкоуровневой части наиболее интересного обобщённого кода, точно так же, как ассемблерный код находится в низкоуровневой части наиболее интересного обычного кода. Как и в ассемблерном коде, `requires`-выражения не должны быть видны в обычном коде. Они относятся к реализации абстракций. Если вы видите `requires requires` в своем коде, это, вероятно, слишком низкий уровень и в конечном итоге это станет проблемой.

Намеренное использование `requires requires` в `advance()` неэлегантно и халтурно. Обратите внимание, что я “забыл” указать `+=` и требуемые возвращаемые типы для операций. Следовательно, некоторые варианты использования версии `advance()` пройдут проверку концепта и все равно не будут компилироваться. Вы были предупреждены! Надлежащая версия `advance()` с произвольным доступом проще и удобочитаемее:

```
template<random_access_iterator Iter>
void advance(Iter p, int n)      // move p n elements forward
{
    p+=n;                        // a random-access iterator has +=
}
```

Предпочтительнее использовать правильно названные концепты с четко определенной семантикой (§8.2.4) и в первую очередь используйте `requires`-выражения в их определении.

8.2.4 Определение концептов

Мы находим полезные концепты, такие как `forward_iterator`, в библиотеках, включая стандартную библиотеку (§14.5). Что касается классов и функций, обычно проще использовать концепт из хорошей библиотеки, чем писать новый, но определить про-

стые концепты несложно. Имена из стандартной библиотеки, такие как `random_access_iterator` и `vector`, написаны строчными буквами. Здесь я использую соглашение для написания с заглавной буквы названий понятий, которые я определил сам, таких как `Sequence` и `Vector`.

Концепт - это предикат времени компиляции, определяющий, как можно использовать один или несколько типов. Рассмотрим сначала один из простейших примеров:

```
template<typename T>
concept Equality_comparable =
    requires (T a, T b) {
        { a == b } -> Boolean;           // compare Ts with ==
        { a != b } -> Boolean;           // compare Ts with !=
    };

```

`Equality_comparable` - это концепт, который мы используем, чтобы гарантировать, что мы можем сравнивать значения типа на равенство и неравенство. Мы просто говорим, что, два значения одного типа, должны быть сопоставимы с помощью `==` и `!=`, и результат этих операций должен быть логическим. Например:

```
static_assert(Equality_comparable<int>);    // succeeds

struct S { int a; };
static_assert(Equality_comparable<S>);      // fails because structs don't automati-
                                           // cally
                                           // get == and !=

```

Определение концепта `Equality_comparable` в точности эквивалентно английскому описанию и не более того. Значение `concept` всегда имеет тип `bool`.

Результат действия `{ ... }`, указанный после `->`, должен быть концептом. К сожалению, `boolean` концепта стандартной библиотеки не существует, поэтому я определил ее (§14.5). `Boolean` просто означает тип, который может быть использован в качестве условия.

Определить `Equality_comparable` для обработки сравнений разных типов почти так же просто:

```
template<typename T, typename T2 = T>
concept Equality_comparable =
    requires (T a, T2 b) {
        { a == b } -> Boolean;           // compare a T to a T2 with ==
        { a != b } -> Boolean;           // compare a T to a T2 with !=
        { b == a } -> Boolean;           // compare a T2 to a T with ==
        { b != a } -> Boolean;           // compare a T2 to a T with !=
    };

```

`typename T2 = T` говорит о том, что если мы не укажем второй аргумент шаблона, `T2` будет таким же, как `T`; `T` - аргумент шаблона по умолчанию.

Мы можем протестировать `Equality_comparable` следующим образом:

```
static_assert(Equality_comparable<int,double>);    // succeeds
static_assert(Equality_comparable<int>);           // succeeds (T2 is defaulted to int)
static_assert(Equality_comparable<int,string>);    // fails

```

`Equality_comparable` почти идентично `equality_comparable` стандартной библиотеки (§14.5).

Теперь мы можем определить концепт, который требует, чтобы между числами были допустимы арифметические операции. Сначала нам нужно определить `Number`:

```
template<typename T, typename U = T>
concept Number =
    requires(T x, U y) { // Something with arithmetic operations and a zero

```

```

    x+y; x-y; x*y; x/y;
    x+=y; x-=y; x*=y; x/=y;
    x=x;           // copy
    x=0;
};

```

Это не делает никаких предположений о типах результатов, но этого достаточно для простого использования. Принимая один тип аргумента, `Number<X>` проверяет, обладает ли `X` желаемыми свойствами числа `Number`. Принимая два аргумента, `Number<X,Y>` проверяет, что эти два типа могут использоваться вместе с требуемыми операциями. Исходя из этого, мы можем определить наш концепт `Arithmetic` (§8.2.1):

```

template<typename T, typename U = T>
concept Arithmetic = Number<T,U> && Number<U,T>;

```

В качестве более сложного примера рассмотрим `Sequence`:

```

template<typename S>
concept Sequence = requires (S a) {
    typename range_value_t<S>;           // S must have a value type
    typename iterator_t<S>;              // S must have an iterator type

    { a.begin() } -> same_as<iterator_t<S>>; // S must have a begin() that re-
turns                                     // an iterator

    { a.end() } -> same_as<iterator_t<S>>;

    requires input_iterator<iterator_t<S>>; // S's iterator must be an
                                           // input_iterator
    requires same_as<range_value_t<S>, iter_value_t<S>>;
};

```

Чтобы тип `S` был `Sequence`, он должен предоставлять тип значение (тип его элементов; см. §13.1) и тип итератор (тип его итераторов). Здесь я использовал соответствующие типы стандартной библиотеки `range_value_t<S>` и `iterator_t<S>` (§16.4.4), чтобы выразить это. Он также должен гарантировать, что существуют функции `begin()` и `end()`, которые возвращают итераторы `S`, что является идиоматичным для контейнеров стандартной библиотеки (§12.3). Наконец, тип итератора `S` должен быть, по крайней мере, `input_iterator`, а типы значений элементов и итератора должны совпадать.

Самые трудные для определения понятия - это те, которые представляют фундаментальные языковые концепции. Следовательно, лучше всего использовать набор из установленной библиотеки. Полезную подборку смотрите в §14.5. В частности, существует концепт стандартной библиотеки, который позволяет нам обойти сложность определения `Sequence`:

```

template<typename S>
concept Sequence = input_range<S>; // simple to write and general

```

Если бы я ограничил свое описание “`S` является типом значения для `S::value_type`, я мог бы использовать простой `Value_type`”:

```

template<class S>
using Value_type = typename S::value_type;

```

Это полезный прием для краткого выражения простых понятий и сокрытия сложности. Определение стандартного `value_type_t` в принципе аналогично, но немного сложнее, поскольку оно обрабатывает последовательности, у которых нет элемента с именем `value_type` (например, встроенные массивы).

8.2.4.1 Проверка определения

Концепты, указанные для шаблона, используются для проверки аргументов в момент использования шаблона. Они *не* используются для проверки использования параметров в определении шаблона. Например:

```
template<equality_comparable T>
bool cmp(T a, T b)
{
    return a<b;
}
```

Здесь концепт гарантирует наличие `==`, но не `<`:

```
bool b0 = cmp(cout, cerr);           // error: ostream doesn't support ==
bool b1 = cmp(2, 3);                 // OK: returns true
bool b2 = cmp(2+3i, 3+4i);           // error: complex<double> doesn't support <
```

Проверка концептов отлавливает попытку передачи `ostream`, но принимает `int` и `complex<double>`, потому что эти два типа поддерживают `==`. Однако `int` поддерживает `<`, поэтому `cmp(2,3)` компилируется, тогда как `cmp(2+3i, 3+4i)` отклоняется, когда тело `cmp()` проверяется и создается экземпляр для `complex<double>`, который не поддерживает `<`.

Отсрочка окончательной проверки определения шаблона до момента создания экземпляра дает два преимущества:

- Мы можем использовать неполные концепты во время разработки. Это позволяет нам набираться опыта при разработке концептов, типов и алгоритмов, а также постепенно совершенствовать проверку.
- Мы можем вставить код отладки, трассировки, телеметрии и т.д. в шаблон, не затрагивая его интерфейс. Изменение интерфейса может привести к массовой перекомпиляции.

И то, и другое важно при разработке и обслуживании больших баз кода. Цена, которую мы платим за это важное преимущество, которое заключается в том, что некоторые ошибки, такие как использование `<`, там где гарантируется только `==`, обнаруживаются очень поздно в процессе компиляции (§8.5).

8.2.5 Концепты и `auto`

Ключевое слово `auto` может использоваться для указания того, что объект должен иметь тип своего инициализатора (§1.4.2):

```
auto x = 1;                          // x is an int
auto z = complex<double>{1,2};       // z is a complex<double>
```

Однако инициализация происходит не только в простых определениях переменных:

```
auto g() { return 99; }               // g() returns an int

int f(auto x) { /* ... */ }           // take an argument of any type

int x = f(1);                         // this f() takes an int
int z = f(complex<double>{1,2});      // this f() takes a complex<double>
```

Ключевое слово `auto` обозначает наименее ограниченную концепцию для значения: оно просто требует, чтобы это было значение некоторого типа. Использование `auto` параметра превращает функцию в шаблон функции.

Учитывая концепции, мы можем усилить требования ко всем таким инициализациям, предваряя `auto` концептом. Например:

```
auto twice(Arithmetic auto x) { return x+x; } // just for numbers
auto thrice(auto x) { return x+x+x; }         // for anything with a +

auto x1 = twice(7);           // OK: x1==14
string s "Hello ";
auto x2 = twice(s);           // error: a string is not Arithmetic
auto x3 = thrice(s);          // OK x3=="Hello Hello Hello "
```

В дополнение к их использованию для ограничения аргументов функций, концепты могут ограничивать инициализацию переменных:

```
auto ch1 = open_channel("foo");           // works with whatever open_channel() re-
turns
Arithmetic auto ch2 = open_channel("foo"); // error: a channel is not Arithmetic
Channel auto ch3 = open_channel("foo");    // OK: assuming Channel is an appropriate
one                                         // concept and that open_channel() returns
```

Это очень удобно для противодействия чрезмерному использованию `auto` и документирования требований к коду с использованием универсальных функций.

Для удобства чтения и отладки часто важно, чтобы ошибка типа была обнаружена как можно ближе к ее источнику. Ограничение возвращаемого типа может помочь:

```
Number auto some_function(int x)
{
    // ...
    return fct(x); // an error unless fct(x) returns a Number
    // ...
}
```

Естественно, мы могли бы достичь этого, введя локальную переменную:

```
auto some_function(int x)
{
    // ...
    Number auto y = fct(x); // an error unless fct(x) returns a Number
    return y;
    // ...
}
```

Однако это немного многословно, и не все типы можно дешево скопировать.

8.2.6 Концепты и типы

Тип:

- Определяет набор операций, которые могут быть применены к объекту явно и неявно
- Опирается на объявления функций и языковые правила
- Определяет, как объект размещается в памяти

Концепт с одним аргументом:

- Определяет набор операций, которые могут быть применены к объекту явно и неявно
- Опирается на шаблоны использования, отражающие объявления функций и языковые правила

- Ничего не говорит о расположении объекта
- Позволяет использовать набор типов

Таким образом, ограничение кода концептами дает большую гибкость, чем ограничение типами. Кроме того, концепты могут определять взаимосвязь между несколькими аргументами. Мой идеал заключается в том, что в конечном итоге большинство функций будут определены как шаблонные функции с их аргументами, ограниченными концептами. К сожалению, нотационная поддержка для этого еще не идеальна: мы должны использовать концепт как прилагательное, а не как существительное. Например:

```
void sort(Sortable auto&);           // 'auto' required
void sort(Sortable&);               // error: 'auto' required after concept name
```

8.3 Обобщённое программирование

Форма *обобщённого программирования*, непосредственно поддерживаемая C++, сосредоточена вокруг идеи абстрагирования от конкретных эффективных алгоритмов для получения обобщённых алгоритмов, которые можно комбинировать с различными представлениями данных для создания широкого спектра полезного программного обеспечения [Stepanov,2009]. Абстракции, которые представляют фундаментальные операции и структуры данных, называются *концептами*.

8.3.1 Использование концептов

Хорошие, полезные концепты являются фундаментальными, и их чаще обнаруживают, чем разрабатывают. Примерами являются целое число и число с плавающей запятой (как определено даже в Classic C [Kernighan,1978]), последовательность и более общие математические понятия, такие как поле и векторное пространство. Они представляют собой фундаментальные концепты той или иной области применения. Вот почему они называются “концептами”. Определение и формализация концептов в той степени, в какой это необходимо для эффективного обобщённого программирования, может оказаться непростой задачей.

Для начала, рассмотрим концепт **regular** (§14.5). Тип является обычным (**regular**), когда он ведет себя очень подобно **int** или **vector**. Объект обычного типа:

- может быть создан конструктором по умолчанию;
- может быть скопирован (с обычной семантикой копирования, в результате чего получаются два объекта, которые независимы и равны при сравнении) с помощью конструктора или присваивания;
- можно сравнить с помощью **==** и **!=**;
- не страдает от технических проблем из-за чрезмерно хитроумных программных трюков.

string - это еще один пример обычного типа **regular**. Как **int**, **string** также является **totally_ordered** (§14.5). То есть две строки можно сравнить с помощью **<**, **<=**, **>**, **>=**, и **<=** с соответствующей семантикой.

Концепт - это не просто синтаксическое понятие, оно в корне связано с семантикой. Например, не определяйте **+** для деления; это не соответствует требованиям для любого разумного числа. К сожалению, у нас пока нет какой-либо языковой поддержки для выражения семантики, поэтому чтобы получить семантически значимые концеп-

ции, нам приходится полагаться на экспертные знания и здравый смысл. Не определяйте семантически бессмысленные понятия, такие как **Addable** (добавляемое) и **Subtractable** (вычитаемое). Вместо этого опирайтесь на знания предметной области для определения концептов, которые соответствуют фундаментальным понятиям в предметной области.

8.3.2 Абстракции использующие шаблоны

Хорошие абстракции бережно выращиваются на конкретных примерах. Не очень хорошая идея пытаться “абстрагироваться”, пытаясь предусмотреть все возможные потребности и методы; в этом направлении кроется неэлегантность и раздувание кода. Вместо этого начните с одного, а лучше с нескольких, конкретных примеров из реального использования и постарайтесь исключить несущественные детали. Рассмотрим:

```
double sum(const vector<int>& v)
{
    double res = 0;
    for (auto x : v)
        res += x;
    return res;
}
```

Очевидно, что это один из многих способов вычисления суммы последовательности чисел.

Рассмотрим, что делает этот код менее общим, чем он должен быть:

- Почему только **int**?
- Почему именно **vector**?
- Почему накапливание суммы в **double**?
- Зачем начинать с **0**?
- Зачем суммируем?

Отвечая на первые четыре вопроса путем преобразования конкретных типов в шаблонные аргументы, мы получаем простейшую форму алгоритма **accumulate** из стандартной библиотеки:

```
template<forward_iterator Iter, Arithmetic<iter_value_t<Iter>> Val>
Val accumulate(Iter first, Iter last, Val res)
{
    for (auto p = first; p!=last; ++p)
        res += *p;
    return res;
}
```

Здесь у нас есть:

- Структура данных, подлежащая обходу, была абстрагирована в пару итераторов, представляющих последовательность (§8.2.4, §13.1).
- Тип аккумулятора был преобразован в параметр.
- Тип аккумулятора должен быть арифметическим.
- Тип накопителя должен работать с типом значения итератора (типом элемента последовательности).
- Начальное значение теперь является входным; тип аккумулятора соответствует типу этого начального значения.

Быстрый анализ или, что еще лучше, измерение покажет, что код, сгенерированный для вызовов с различными структурами данных, идентичен тому, что вы получаете из примеров, написанных вручную. Рассмотрим:

```
void use(const vector<int>& vec, const list<double>& lst)
{
    auto sum = accumulate(begin(vec),end(vec),0.0); // accumulate in a double
    auto sum2 = accumulate(begin(lst),end(lst),sum);
    // ...
}
```

Процесс обобщения из конкретного фрагмента кода (и предпочтительнее из нескольких) при сохранении производительности называется *лифтингом* (*lifting*). И наоборот, лучший способ разработать шаблон часто заключается в том, чтобы

- сначала написать конкретную версию
- затем отладить, протестировать и измерить её
- наконец, заменить конкретные типы шаблонными аргументами.

Естественно, повторение `begin()` и `end()` утомительно, поэтому мы можем немного упростить пользовательский интерфейс:

```
template<forward_range R, Arithmetic<value_type_t<R>> Val>
Val accumulate(const R& r, Val res = 0)
{
    for (auto x : r)
        res += x;
    return res;
}
```

Диапазон - это концепция стандартной библиотеки, представляющая последовательность с `begin()` и `end()` (§13.1). Для полной обобщённости мы также можем абстрагировать операцию `+=`; см. §17.3.

Полезны как пара итераторов, так и версия `accumulate()` для диапазона: версия пары итераторов для универсальности, версия для диапазона для простоты общего использования.

8.4 Шаблоны с переменным числом аргументов

Шаблон может быть определен таким образом, чтобы принимать произвольное количество аргументов произвольных типов. Такой шаблон называется *вариадик шаблоном* или *шаблоном с переменным количеством аргументов* (*variadic template*). Рассмотрим простую функцию для записи значений любого типа, которая имеет оператор `<<`:

```
void user()
{
    print("first: ", 1, 2.2, "hello\n"s); // first: 1 2.2 hello
    print("\nsecond: ", 0.2, 'c', "yuck!"s, 0, 1, 2, '\n'); // second: 0.2 c yuck! 0
1 2
}
```

Традиционно реализация вариадик шаблона заключалась в том, чтобы отделить первый аргумент от остальных, а затем рекурсивно вызвать вариадик шаблон для конца аргументов:

```
template<typename T>
concept Printable = requires(T t) { std::cout << t; } // just one operation!

void print()
```

```

{
    // what we do for no arguments: nothing
}

template<Printable T, Printable... Tail>
void print(T head, Tail... tail)
{
    cout << head << ' ';           // first, what we do for the head
    print(tail...);                 // then, what we do for the tail
}

```

`Printable...` указывает, что `Tail` - это последовательность типов. `Tail...` указывает, что `tail` - это последовательность значений типов в `Tail`. Параметр, объявленный с помощью `...`, называется *пакетом параметров*. Здесь `tail` (аргумент функции) - это пакет параметров, в котором элементы относятся к типам, найденным в пакете параметров `Tail` (аргумент шаблона). Итак, `print()` может принимать любое количество аргументов любых типов.

Вызов `print()` разделяет аргументы на голову `head` (первый) и хвост `tail` (остальные). Печатается голова, а затем вызывается функция `print()` для хвоста. В конце концов, конечно, `tail` станет пустым, поэтому нам нужна версия `print()` без аргументов, чтобы справиться с этим. Если мы не хотим допускать случай с нулевым аргументом, мы можем исключить эту функцию `print()`, используя `if` времени компиляции:

```

template<Printable T, Printable... Tail>
void print(T head, Tail... tail)
{
    cout << head << ' ';
    if constexpr(sizeof...(tail)> 0)
        print(tail...);
}

```

Я использовал `if` времени компиляции (§7.4.3), а не обычный `if` времени выполнения, чтобы избежать генерации окончательного вызова `print()`. Учитывая это, “пустую” функцию `print()` определять не нужно.

Сила вариадик шаблонов заключается в том, что они могут принимать любые аргументы, которые вы пожелаете им предоставить. Слабые стороны включают в себя

- Рекурсивные реализации может быть сложно реализовать правильно.
- Проверка типов интерфейса – это, возможно, сложно разрабатываемый шаблонный код.
- Код проверки типа является специальным, а не определенным в стандарте.
- Рекурсивные реализации могут быть на удивление дорогостоящими с точки зрения времени компиляции и требований к памяти компилятора.

Из-за своей гибкости вариадик шаблоны широко используются в стандартной библиотеке, а иногда даже чрезмерно широко.

8.4.1 Выражения свёртки

Чтобы упростить реализацию простых вариадик шаблонов, C++ предлагает ограниченную форму итерации по элементам пакета параметров. Например:

```

template<Number... T>
int sum(T... v)
{
    return (v + ... + 0);           // add all elements of v starting with 0
}

```

Эта функция `sum()` может принимать любое количество аргументов любых типов:

```
int x = sum(1, 2, 3, 4, 5); // x becomes 15
int y = sum('a', 2.4, x); // y becomes 114 (2.4 is truncated and the value of 'a' is 97)
```

В теле `sum` используется выражение свёртки:

```
return (v + ... + 0); // add all elements of v to 0
```

Здесь `(v+...+0)` означает сложение всех элементов `v`, начиная с начального значения `0`. Первым добавляемым элементом является “самый правый” (с наибольшим индексом): `(v[0]+(v[1]+(v[2]+(v[3]+(v[4]+0)))))`. То есть, начиная справа, где находится `0`. Это называется *правой свёрткой*. В качестве альтернативы мы могли бы использовать *левую свёртку*:

```
template<Number... T>
int sum2(T... v)
{
    return (0 + ... + v); // add all elements of v to 0
}
```

Теперь первым добавляемым элементом является “крайний левый” (с наименьшим индексом): `(((((0+v[0])+v[1])+v[2])+v[3])+v[4])`. То есть, начиная слева, где находится `0`.

Свёртка (fold) - это очень мощная абстракция, явно связанная с `accumulate()` стандартной библиотеки, с множеством названий в разных языках и сообществах. В C++ свёртка выражений в настоящее время ограничена для упрощения реализации вариадик шаблонов. Свёртке не обязательно выполнять числовые вычисления. Рассмотрим известный пример:

```
template<Printable ...T>
void print(T&&... args)
{
    (std::cout << ... << args) << '\n'; // print all arguments
}

print("Hello!"s, ' ', "World ", 2017); // (((((std::cout << "Hello!"s) << ' ') <<
// "World ") << 2017) << '\n');
```

Почему именно 2017 год? Потому что функция свёртки `fold()` была добавлена в C++ в 2017 году (§19.2.3).

8.4.2 Передача аргументов

Передача аргументов через интерфейс в неизменном виде является важным случаем использованием вариадик шаблонов. Рассмотрим понятие сетевого входного канала, для которого фактический метод перемещения значений является параметром. Различные транспортные механизмы имеют разные наборы параметров конструктора:

```
template<concepts::InputTransport Transport>
class InputChannel {
public:
    // ...
    InputChannel(Transport::Args&&... transportArgs)
        : _transport(std::forward<TransportArgs>(transportArgs)...)
    {}
    // ...
    Transport _transport;
};
```

Функция `forward()` стандартной библиотеки (§16.6) используется для передачи аргументов без изменений из конструктора `InputChannel` в конструктор `Transport`.

Дело в том, что автор `InputChannel` может создать объект типа `Transport` без необходимости знать, какие аргументы требуются для построения конкретного `Transport`. Разработчику `InputChannel` необходимо только знать общий пользовательский интерфейс для всех `Transport` объектов.

Пересылка очень распространена в базовых библиотеках, где необходимы универсальность и низкие накладные расходы во время выполнения, а также распространены очень общие интерфейсы.

8.5 Модель компиляции шаблонов

В момент обращения аргументы шаблона проверяются на соответствие его концептам. Об обнаруженных здесь ошибках будет немедленно сообщено. То, что не может быть проверено на данном этапе, например, аргументы для неограниченных параметров шаблона, откладывается до тех пор, пока для шаблона не будет сгенерирован код с набором аргументов шаблона: “во время создания экземпляра шаблона”.

Неприятным побочным эффектом проверки типа во время создания экземпляра является то, что ошибка типа может быть обнаружена с неприятной задержкой (§8.2.4.1). Кроме того, поздняя проверка часто приводит к появлению впечатляюще плохих сообщений об ошибках, поскольку компилятор не располагает информацией о типе, дающей подсказки о намерениях программиста, и часто обнаруживает проблему только после объединения информации из нескольких мест в программе.

Проверка типа во время создания экземпляра, предусмотренная для шаблонов, проверяет использование аргументов в определении шаблона. Это обеспечивает во время компиляции вариант того, что часто называют *утиной типизацией* (“Если нечто выглядит как утка, ходит как утка и кричит как утка, то это утка”). Или – используя более техническую терминологию – мы оперируем значениями, а наличие и значение операции зависят исключительно от значений операндов. Это отличается от альтернативного представления о том, что объекты имеют типы, которые определяют наличие и значение операций. Значения “живут” в объектах. Именно так объекты (например, переменные) работают в C++, и в него могут быть помещены только значения, соответствующие требованиям объекта. То, что делается во время компиляции с использованием шаблонов, в основном не связано с объектами, только со значениями. Исключением являются локальные переменные в `constexpr` функции (§1.6), которые используются как объекты внутри компилятора.

Чтобы использовать неограниченный шаблон, его определение (а не только его объявление) должно находиться в области видимости в месте его использования. При использовании файлов заголовков и `#include` это означает, что определения шаблонов должны находиться в заголовочных файлах, а не в файлах `.cpp`. Например, стандартный заголовок `<vector>` содержит определение `vector`.

Это меняется, когда мы начинаем использовать модули (§3.2.2). Используя модули, исходный код может быть организован для шаблонных функций таким же образом, как и для обычных функций. Модуль частично компилируется в представление, что позволяет быстро импортировать `import` и использовать его. Думайте об этом представлении как о легко проходимом графе, содержащем всю доступную информацию о области и типе и поддерживаемом таблицей символов, позволяющей быстро получить доступ к отдельным объектам.

8.6 Советы

- [1] Шаблоны предоставляют общий механизм для программирования времени компиляции; §8.1.
- [2] При разработке шаблона внимательно продумывайте концепты (требования), предъявляемые к аргументам шаблона; §8.3.2.
- [3] При разработке шаблона используйте конкретную версию для первоначальной реализации, отладки и измерения; §8.3.2.
- [4] Используйте концепты в качестве инструмента проектирования; §8.2.1.
- [5] Указывайте концепты для всех аргументов шаблона; §8.2; [CG: T.10].
- [6] По возможности используйте именованные концепты (например, концепты стандартной библиотеки); §8.2.4, §14.5; [CG: T.11].
- [7] Используйте шаблоны для описания контейнеров и диапазонов; §8.3.2; [CG: T.3].
- [8] Избегайте “концепты” без значимой семантики; §8.2; [CG: T.20].
- [9] Требуйте полный набор операций в концепте; §8.2; [CG: T.21].
- [10] Используйте именованные концепты §8.2.3.
- [11] Избегайте `requires requires`; §8.2.3.
- [12] `auto` - концепт с наименьшими ограничениями §8.2.5.
- [13] Используйте вариадик шаблоны, когда вам нужна функция, которая принимает переменное количество аргументов различных типов; §8.4.
- [14] Шаблоны используют “утиную типизацию” времени компиляции; §8.5.
- [15] При использовании заголовочных файлов, включайте `#include` определения шаблонов (а не только объявления) в каждую единицу трансляции, которая их использует; §8.5.
- [16] При использовании шаблона, убедитесь, что его определение (а не только объявление) находится в области видимости; §8.5.
- [17] Неограниченные шаблоны предлагают “утиный ввод” времени компиляции; §8.5.

Обзор стандартной библиотеки

Зачем тратить время на обучение, когда невежество приходит мгновенно?
– Hobbes

- [Введение](#)
- [Компоненты стандартной библиотеки](#)
- [Организация стандартной библиотеки](#)

[Пространства имён](#); [Пространство имён `ranges`](#); [Модули](#); [Заголовочные файлы](#)

- [Советы](#)

9.1 Введение

Ни одна серьёзная программа не написана на голом языке программирования. Во-первых, имеется набор уже существующих библиотек, которые формируют основу для дальнейшей разработки. Большинство программ утомительно писать на голом языке, тем более практически любая задача может быть упрощена с помощью хороших библиотек.

Продолжая [главы 1-8](#), в [главах 9-18](#) дается краткий обзор ключевых возможностей стандартной библиотеки. Я очень кратко представляю полезные типы стандартной библиотеки, такие как `string`, `ostream`, `variant`, `vector`, `map`, `path`, `unique_ptr`, `thread`, `regex`, `system_clock`, `time_zone` и `complex`, а также наиболее распространенные способы их применения.

Как и в [главах 1-8](#), вам настоятельно рекомендуется не отвлекаться и не расстраиваться из-за неполного понимания деталей. Цель этой главы - дать общее представление о наиболее полезных библиотечных средствах.

Спецификация стандартной библиотеки составляет более двух третей от стандарта ISO C++. Изучите её и используйте вместо самописных «велосипедов». Много умственного труда было вложено в разработку стандартной библиотеки, еще больше - в её реализацию, и много усилий будет затрачено на её обслуживание и расширение.

Средства стандартной библиотеки, описанные в этой книге, являются частью каждой полной реализации C++. В дополнение к компонентам стандартной библиотеки, большинство реализаций предлагают системы “графического пользовательского интерфейса” (GUI), веб-интерфейсы, интерфейсы баз данных и т.д. Аналогичным образом, большинство сред разработки приложений предоставляют “базовые библиотеки” для корпоративных или промышленных “стандартных” сред разработки и/или выполне-

ния. Помимо этого, существует не одна тысяча библиотек, поддерживающих специализированные области применения. Здесь я не описываю библиотеки, системы или среды, выходящие за рамки стандартной библиотеки. Цель состоит в том, чтобы предоставить самодостаточное описание C++, определённое в стандарте [C++,2020], и сохранить переносимость примеров. Естественно, программисту рекомендуется изучить более обширные возможности, доступные в большинстве систем.

9.2 Компоненты стандартной библиотеки

Средства, предоставляемые стандартной библиотекой, могут быть классифицированы следующим образом:

- Поддержка языком программирования времени выполнения (например, для аллокации, исключений и информации о типе во время выполнения RTTI)
- Стандартная библиотека C (с очень незначительными изменениями, чтобы свести к минимуму нарушения системы типов).
- Строки с поддержкой международных наборов символов, локализации и представления подстрок только для чтения (§10.2).
- Поддержка регулярных выражений (§10.4).
- Потоки ввода-вывода - это расширяемая платформа для ввода и вывода, в которую пользователи могут добавлять свои собственные типы, потоки, стратегии буферизации, локали и наборы символов (глава 11). Он также предлагает средства для гибкого форматирования выходных данных (§11.6.2).
- Кроссплатформенная библиотека для управления файловыми системами (§11.9).
- Базовый набор контейнеров (таких как `vector` и `map`; глава 12) и алгоритмов (таких как `find()`, `sort()` и `merge()`; глава 13). Этот набор, условно называемый STL [Stepanov,1994], является расширяемым, поэтому пользователи могут добавлять свои собственные контейнеры и алгоритмы.
- Диапазоны (§14.1), включая представления (§14.2), генераторы (§14.3) и конвейеры (§14.4).
- Концепты для основных типов и диапазонов (§14.5).
- Поддержка числовых вычислений, таких как стандартные математические функции, комплексные числа, векторы с арифметическими операциями, математические константы и генераторы случайных чисел (§5.2.1 и глава 16).
- Поддержка параллельного программирования, включая потоки `thread` и блокировки `lock` (глава 18). Поддержка параллелизма является основополагающей, так что пользователи могут добавлять поддержку новых моделей параллелизма в виде библиотек.
- Синхронные и асинхронные сопрограммы (§18.6).
- Параллельные версии большинства алгоритмов STL и некоторых численных алгоритмов, таких как `sort()` (§13.6) и `reduce()` (§17.3.1).
- Утилиты для поддержки метапрограммирования (например, функции типов; §16.4), универсального программирования в стиле STL (например, `pair`; §15.3.3) и обобщённого программирования (например, `variant` и `optional`; §15.4.1, §15.4.2).

- “Умные указатели” для управления ресурсами (например, `unique_ptr` и `shared_ptr`; §15.2.1).
- Контейнеры специального назначения, такие как `array` (§15.3.1), `bitset` (§15.3.2) и `tuple` (§15.3.3).
- Поддержка абсолютного времени и длительностей, например, `time_point` и `system_clock` (§16.2.1).
- Поддержка календарей, например, `month` и `time_zone` (§16.2.2, §16.2.3).
- Суффиксы для популярных единиц измерения, таких как `ms` для миллисекунд и `i` для мнимых чисел (§6.6).
- Способы манипулирования последовательностями элементов, такими как представления (§14.2), `string_view` (§10.3) и `span` (§15.2.2).

Основными критериями для включения класса в библиотеку были следующие:

- Он может быть полезен почти каждому программисту на C++ (как новичкам, так и экспертам).
- Он мог бы быть предоставлен в общем виде, который не добавляет значительных накладных расходов по сравнению с более простой версией того же средства.
- Простые способы использования должны быть легкими в освоении (относительно сложности, присущей им задачи).

По сути, стандартная библиотека C++ предоставляет наиболее распространенные фундаментальные структуры данных вместе с фундаментальными алгоритмами, используемыми в них.

9.3 Организация стандартной библиотеки

Средства стандартной библиотеки размещены в пространстве имен `std` и доступны пользователям через модули или заголовочные файлы.

9.3.1 Пространства имён

Каждое средство стандартной библиотеки предоставляется через некоторый стандартный заголовочный файл. Например:

```
#include<string>
#include<list>
```

Эта запись делает доступными стандартные `string` и `list`.

Стандартная библиотека определена в пространстве имен (§3.3), называемом `std`. Для использования средств стандартной библиотеки можно использовать префикс `std::`:

```
std::string sheep {"Four legs Good; two legs Baaad!"};
std::list<std::string> slogans {"War is Peace", "Freedom is Slavery", "Ignorance is Strength"};
```

Для краткости я редко использую префикс `std::` в примерах. Я также не включаю `#include` и не импортирую `import` необходимые заголовки или модули явно. Чтобы скомпилировать и запустить приведенные здесь фрагменты программы, вы должны сделать доступными соответствующие части стандартной библиотеки. Например:

```
#include<string>                // make the standard string facilities accessible
using namespace std;           // make std names available without std:: prefix

string s {"C++ is a general-purpose programming language"}; // OK: string is
std::string
```

Как правило, это дурной тон - сбрасывать каждое имя из пространства имён в глобальное пространство имён. Однако в этой книге я использую исключительно стандартную библиотеку, и полезно знать, что она предлагает.

Стандартная библиотека предлагает несколько вложенных пространств имен для **std**, доступ к которым возможен только с помощью явного указания:

- **std::chrono**: все удобства от chrono, включая **std::literals::chrono_literals** (§16.2).
- **std::literals::chrono_literals**: суффиксы обозначающие: **y** - годы, **d** - дни, **h** - часы, **min** - минуты, **ms** - миллисекунды, **ns** - наносекунды, **s** - секунды и **us** - микросекунды (§16.2).
- **std::literals::complex_literals**: суффиксы **i** для мнимых double, **if** для мнимых float и **il** для мнимых long double (§6.6).
- **std::literals::string_literals**: суффикс **s** для строк (§6.6, §10.2).
- **std::literals::string_view_literals**: суффикс **sv** для строкового представления (§10.3).
- **std::numbers** для математических констант (§17.9).
- **std::pmr** для полиморфных ресурсов памяти (§12.7).

Чтобы использовать суффикс из подпространства имен, мы должны ввести его в пространство имен, в котором мы хотим его использовать. Например:

```
// no mention of complex_literals
auto z1 = 2+3i;           // error: no suffix 'i'

using namespace literals::complex_literals; // make the complex literals visible
auto z2 = 2+3i;           // ok: z2 is a complex<double>
```

Не существует последовательной философии для того, что должно быть в подпространстве имен. Однако суффиксы не могут быть определены явно, поэтому мы можем ввести в область видимости только один набор суффиксов, не рискуя вызвать двусмысленности. Поэтому суффиксы для библиотеки, предназначенной для работы с другими библиотеками (которые могут определять свои собственные суффиксы), помещаются во вложенные пространства имен.

9.3.2 Пространство имён **ranges**

Стандартная библиотека предлагает алгоритмы, такие как **sort()** и **copy()**, в двух версиях:

- Традиционная версия последовательности, использующая пару итераторов; например, **sort(begin(v), v.end())**
- Версия диапазона, использующая один диапазон; например, **sort(v)**

В идеале эти две версии должны прекрасно перегружаться без каких-либо особых усилий. Однако они этого не делают. Например:

```
using namespace std;
using namespace ranges;

void f(vector<int>& v)
{
```

```

    sort(v.begin(),v.end());    // error: ambiguous
    sort(v);                    // error: ambiguous
}

```

Для защиты от двусмысленностей при использовании традиционных неограниченных шаблонов стандарт требует, чтобы мы явно вводили версию диапазона алгоритма стандартной библиотеки в область видимости:

```

using namespace std;

void g(vector<int>& v)
{
    sort(v.begin(),v.end());    // OK
    sort(v);                    // error: no matching function (in std)
    ranges::sort(v);            // OK
    using ranges::sort;         // sort(v) OK from here on
    sort(v);                    // OK
}

```

9.3.3 Модули

Пока нет никаких модулей стандартной библиотеки. C++23, вероятно, исправит это упущение (вызванное нехваткой времени у комитета). На данный момент я использую `module std`, который, вероятно, станет стандартным, предлагая все возможности из `namespace std`. См. Приложение А.

9.3.4 Заголовочные файлы

Вот подборка заголовков стандартной библиотеки, все из которых содержат объявления в пространстве имен `std`:

Избранные заголовочные файлы стандартной библиотеки		
<code><algorithm></code>	<code>copy()</code> , <code>find()</code> , <code>sort()</code>	Chapter 13
<code><array></code>	<code>array</code>	§15.3.1
<code><chrono></code>	<code>duration</code> , <code>time_point</code> , <code>month</code> , <code>time_zone</code>	§16.2
<code><cmath></code>	<code>sqrt()</code> , <code>pow()</code>	§17.2
<code><complex></code>	<code>complex</code> , <code>sqrt()</code> , <code>pow()</code>	§17.4
<code><concepts></code>	<code>floating_point</code> , <code>copyable</code> , <code>predicate</code> , <code>invocable</code>	§14.5
<code><filesystem></code>	<code>path</code>	§11.9
<code><format></code>	<code>format()</code>	§11.6.2
<code><fstream></code>	<code>fstream</code> , <code>ifstream</code> , <code>ofstream</code>	§11.7.2
<code><functional></code>	<code>function</code> , <code>greater_equal</code> , <code>hash</code> , <code>range_value_t</code>	Chapter 16
<code><future></code>	<code>future</code> , <code>promise</code>	§18.5
<code><ios></code>	<code>hex</code> , <code>dec</code> , <code>scientific</code> , <code>fixed</code> , <code>defaultfloat</code>	§11.6.2
<code><iostream></code>	<code>istream</code> , <code>ostream</code> , <code>cin</code> , <code>cout</code>	Chapter 11
<code><map></code>	<code>map</code> , <code>multimap</code>	§12.6
<code><memory></code>	<code>unique_ptr</code> , <code>shared_ptr</code> , <code>allocator</code>	§15.2.1
<code><random></code>	<code>default_random_engine</code> , <code>normal_distribution</code>	§17.5
<code><ranges></code>	<code>sized_range</code> , <code>subrange</code> , <code>take()</code> , <code>split()</code> , <code>iterator_t</code>	§14.1
<code><regex></code>	<code>regex</code> , <code>smatch</code>	§10.4

<code><string></code>	<code>string, basic_string</code>	§10.2
<code><string_view></code>	<code>string_view</code>	§10.3
<code><set></code>	<code>set, multiset</code>	§12.8
<code><sstream></code>	<code>istringstream, ostringstream</code>	§11.7.3
<code><stdexcept></code>	<code>length_error, out_of_range, runtime_error</code>	§4.2
<code><tuple></code>	<code>tuple, get<>(), tuple_size<></code>	§15.3.4
<code><thread></code>	<code>thread</code>	§18.2
<code><unordered_map></code>	<code>unordered_map, unordered_multimap</code>	§12.6
<code><utility></code>	<code>move(), swap(), pair</code>	Chapter 16
<code><variant></code>	<code>variant</code>	§15.4.1
<code><vector></code>	<code>vector</code>	§12.2

Этот список далеко не полный.

Заголовки из стандартной библиотеки C, такие как `<stdlib.h>`, тоже поддерживаются. Для каждого такого заголовка также существует версия с префиксом `c` в названии и удаленным `.h`. Эта версия, такая как `<cstdlib>`, помещает свои объявления как в `std`, так и в глобальное пространство имен.

Заголовки отражают историю разработки стандартной библиотеки. Следовательно, они не всегда так логичны и легки для запоминания, как хотелось бы. Это одна из причин использовать вместо этого модуль, такой как `std` ([§9.3.3](#)).

9.4 Советы

- [1] Не изобретайте велосипед заново; используйте библиотеки; [§9.1](#); [CG: SL.1.]
- [2] Когда у вас есть выбор, отдайте предпочтение стандартной библиотеке перед другими библиотеками; [§9.1](#); [CG: SL.2].
- [3] Не думайте, что стандартная библиотека идеальна для всего; [§9.1](#).
- [4] Если вы не используете модули, не забудьте включить `#include` соответствующие заголовки; [§9.3.1](#).
- [5] Помните, что средства стандартной библиотеки определены в пространстве имен `std`; [§9.3.1](#); [CG: SL.3].
- [6] При использовании диапазонов `ranges` не забудьте явно указать имена алгоритмов; [§9.3.2](#).
- [7] Предпочтительнее импортировать модули `import module`, а не `#include` заголовочные файлы ([§9.3.3](#)).

Строки и регулярные выражения

*Предпочитайте стандартное
необычному.
– Strunk & White*

- [Введение](#)
- [Строки](#)

[string](#) [Реализация](#)

- [Строковые представления](#)
- [Регулярные выражения](#)

[Поиск](#); [Описание регулярного выражения](#); [Итераторы](#)

- [Советы](#)

10.1 Введение

Обработка текста является основной частью большинства программ. Стандартная библиотека C++ предлагает строковый тип `string`, чтобы избавить большинство пользователей от манипулирования массивами символов в стиле C через указатели. Тип `string_view` позволяет нам работать с последовательностями символов, независимо от того, где они хранятся (например, в `std::string` или `char[]`). Кроме того, предлагается сопоставление строк с регулярными выражениями, помогающее находить шаблоны в тексте. Регулярные выражения представлены в форме, аналогичной той, что распространена в большинстве современных языков. Как строки `string`, так и объекты регулярных выражений `regex` могут использовать различные типы символов (например, Unicode).

10.2 Строки

Стандартная библиотека предоставляет тип `string` в дополнение к строковым литералам (§1.2.1); `string` - это обычный (концепт `regular`) тип (§8.2, §14.5) для владения и манипулирования последовательностью символов различных типов. Тип `string` предоставляет множество полезных операций со строками, таких как конкатенация (объединение). Например:

```
string compose(const string& name, const string& domain)
{
    return name + @ + domain;
}

auto addr = compose("dmr", "bell-labs.com");
```

Здесь `addr` инициализируется последовательностью символов `dmr@bell-labs.com`. “Сложение” строк подразумевает их объединение. Вы можете объединить строку, строковый литерал, строку в стиле C или символ со строкой. Стандартная строка имеет конструктор перемещения, поэтому возврат даже длинных строк по значению эффективен (§6.2.2).

Во многих приложениях наиболее распространенной формой конкатенации является добавление чего-либо в конец строки. Это напрямую поддерживается операцией `+=`. Например:

```
void m2(string& s1, string& s2)
{
    s1 = s1 + '\n';    // append newline
    s2 += '\n';        // append newline
}
```

Эти два способа добавления в конец строки семантически эквивалентны, но я предпочитаю последний, потому что он более четко описывает, что он делает, более лаконичен и, возможно, более эффективен.

Строка изменяема. В дополнение к `=` и `+=` поддерживаются операции индекса (с использованием `[]`) и подстроки. Например:

```
string name = "Niels Stroustrup";

void m3()
{
    string s = name.substr(6,10);           // s = "Stroustrup"
    name.replace(0,5,"nicholas");          // name becomes "nicholas Stroustrup"
    name[0] = toupper(name[0]);            // name becomes "Nicholas Stroustrup"
}
```

Операция `substr()` возвращает строку, которая является копией подстроки, указанной в ее аргументах. Первый аргумент - это индекс в строке (позиция), а второй - длина подстроки. Поскольку индексация начинается с 0, `s` принимает значение `Stroustrup`.

Операция `replace()` заменяет подстроку значением. В этом случае подстрокой, начинающейся с 0 и имеющей длину 5, является `Niels`; она заменяется на `nicholas`. Наконец, я заменяю начальный символ его эквивалентом в верхнем регистре. Таким образом, конечное значение `name` - `Nicholas Stroustrup`. Обратите внимание, что заменяющая строка не обязательно должна быть того же размера, что и подстрока, которую она заменяет.

Среди множества полезных операций со строками - присваивание (с использованием `=`), индекс (с использованием `[]` или `at()`, как для `vector`; §12.2.2), сравнение (с использованием `==` и `!=`) и лексикографический порядок (с использованием `<`, `<=`, `>` и `>=`), итерирование (с использованием итераторов, `begin()` и `end()` как для `vector`; §13.2), ввод (§11.3) и потоки (§11.7.3).

Естественно, строки можно сравнивать друг с другом, со строками в стиле C (§1.7.1) и со строковыми литералами. Например:

```
string incantation;

void respond(const string& answer)
{
    if (answer == incantation) {
        // ... perform magic ...
    }
    else if (answer == "yes") {
        // ...
    }
}
```

```

    // ...
}

```

Если вам нужна строка в стиле C (массив символов, заканчивающийся нулем), `string` предлагает доступ только для чтения к содержащимся в нем символам (`c_str()` и `data()`). Например:

```

void print(const string& s)
{
    printf("For people who like printf: %s\n", s.c_str()); // s.c_str() returns a
    // pointer
    cout << "For people who like streams: " << s << '\n'; // to s' characters
}

```

Строковый литерал по умолчанию является `const char*`. Чтобы получить литерал типа `std::string`, используйте суффикс `s`. Например:

```

auto cat = "Cat"s; // a std::string
auto dog = "Dog"; // a C-style string: a const char*

```

Чтобы использовать суффикс `s`, вам необходимо использовать пространство имен `std::literals::string_literals` (§6.6).

10.2.1 Реализация `string`

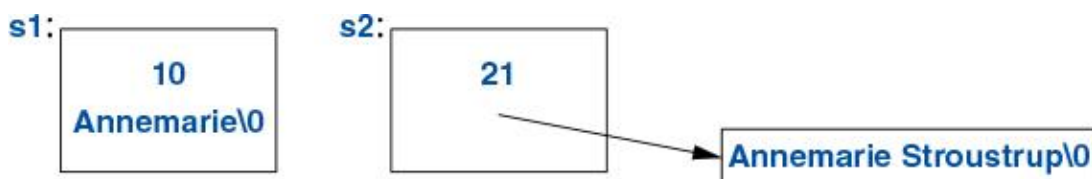
Реализация класса `string` - популярное и полезное упражнение. Однако для широкого применения наши бережно разработанные первые попытки редко соответствуют стандартному `string` по удобству или производительности. В наши дни `string` обычно реализуется с использованием *оптимизации коротких строк*. То есть короткие строковые значения сохраняются в самом объекте `string`, и только более длинные строки помещаются в динамическую память. Рассмотрим:

```

string s1 {"Annemarie"}; // short string
string s2 {"Annemarie Stroustrup"}; // long string

```

Расположение в памяти будет примерно таким:



Когда значение `string` изменяется с короткой на длинную строку (и наоборот), ее представление соответствующим образом корректируется. Сколько символов может содержать “короткая” строка? Это определено реализацией, но “около 14 символов” - неплохое предположение.

Фактическая производительность строк может критически зависеть от среды выполнения. В частности, в многопоточных реализациях выделение памяти может быть относительно дорогостоящим. Кроме того, при использовании большого количества строк разной длины может возникнуть фрагментация памяти. Это основные причины того, что оптимизация коротких строк стала повсеместной.

Для обработки нескольких наборов символов `string` на самом деле является псевдонимом для общего шаблона `basic_string` с символьным типом `char`:

```

template<typename Char>
class basic_string {

```



```
// ... string of Char ...
};

using string = basic_string<char>;
```

Пользователь может определять строки произвольных типов символов. Например, предполагая, что у нас есть японский символ типа **Jchar**, мы можем написать:

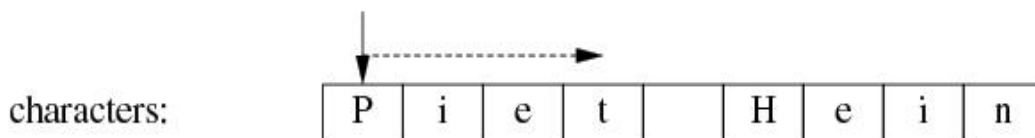
```
using Jstring = basic_string<Jchar>;
```

Теперь мы можем выполнять все обычные строковые операции над **Jstring**, строкой японских символов.

10.3 Строковые представления

Наиболее распространенное использование последовательности символов - это передача ее какой-либо функции для чтения. Это может быть достигнуто путем передачи строки по значению, по ссылке или виде строки в стиле C. Во многих системах существуют дополнительные альтернативы, такие как строковые типы, не предлагаемые стандартом. Во всех этих случаях возникают дополнительные сложности при передаче подстроки. Чтобы решить эту проблему, стандартная библиотека предлагает **string_view**; **string_view** - это, по сути, пара (указатель, длина), обозначающая последовательность символов:

```
string_view: { begin() , size() }
```



string_view предоставляет доступ к непрерывной последовательности символов. Символы могут быть храниться разными способами, в том числе в виде **string** и в виде строки в стиле C. **string_view** подобен указателю или ссылке в том смысле, что ему не принадлежат символы, на которые он указывает. В этом он напоминает пару итераторов STL (§13.3).

Рассмотрим простую функцию, объединяющую две строки:

```
string cat(string_view sv1, string_view sv2)
{
    string res {sv1};           // initialize from sv1
    return res += sv2;          // append from sv2 and return
}
```

Мы можем вызвать функцию **cat()**:

```
string king = "Harold";
auto s1 = cat(king, "William");           // HaroldWilliam: string and const char*
auto s2 = cat(king, king);                 // HaroldHarold: string and string
auto s3 = cat("Edward", "Stephen"sv);      // EdwardStephen: const char * and
string_view

auto s4 = cat("Canute"sv, king);            // CanuteHarold
auto s5 = cat({&king[0],2}, "Henry"sv);    // HaHenry
auto s6 = cat({&king[0],2}, {&king[2],4}); // Harold
```

Функция **cat()** имеет три преимущества перед **compose()**, который принимает **const string&** аргументы (§10.2):

- Его можно использовать для последовательностей символов, управляемых различными способами.
- Мы можем легко передать подстроку.
- Нам не нужно создавать `string` для передачи строкового аргумента в стиле C.

Обратите внимание на использование суффикса `sv` ("string view"). Чтобы использовать его, нам нужно сделать его видимым:

```
using namespace std::literals::string_view_literals;    // §6.6
```

Зачем беспокоиться о суффиксе? Причина в том, что когда мы передаем `"Edward"`, нам нужно создать `string_view` из `const char*`, а это требует подсчета символов. Для `"Stephen"` `sv` длина вычисляется во время компиляции.

`string_view` определяет диапазон, поэтому мы можем перемещаться по его символам. Например:

```
void print_lower(string_view sv1)
{
    for (char ch : sv1)
        cout << tolower(ch);
}
```

Одним из существенных ограничений `string_view` является то, что его символы доступны только для чтения. Например, вы не можете использовать `string_view` для передачи символов функции, которая изменяет свой аргумент на нижний регистр. Для этого вы могли бы рассмотреть возможность использования `span` (§15.2.2).

Думайте о `string_view` как о своего рода указателе; чтобы его использовать, он должен указывать на что-то:

```
string_view bad()
{
    string s = "Once upon a time";
    return {&s[5],4};    // bad: returning a pointer to a local
}
```

Здесь возвращаемый `string` будет уничтожен до того, как мы сможем его использовать.

Поведение при доступе к `string_view` за пределами диапазона не определено. Если вам нужна гарантированная проверка диапазона, используйте `at()`, которая бросит исключение `out_of_range` при попытке доступа за пределы диапазона, или `gsl::string_span` (§15.2.2).

10.4 Регулярные выражения

Регулярные выражения - это мощный инструмент для обработки текста. Они предоставляют способ простого и сжатого описания шаблонов в тексте (например, почтовый индекс США, такой как `TX 77845`, или дату в стиле ISO, такую как `2009-06-07`) и эффективного поиска таких шаблонов. В `<regex>` стандартная библиотека обеспечивает поддержку регулярных выражений в виде класса `std::regex` и его вспомогательных функций. Чтобы получить наглядное представление о библиотеке `regex`, давайте определим и напечатаем шаблон:

```
regex pat {R"(\w{2}\s*\d{5}(-\d{4})?)"};    // U.S. postal code pattern: XXdddd-dddd
and variants
```

Люди, которые использовали регулярные выражения практически на любом языке, найдут `\w{2}\s*\d{5}(-\d{4})?` знакомый. Он задает шаблон, начинающийся с двух букв `\w{2}`,

за которыми необязательно следует некоторый пробел `\s*`, за которым следуют пять цифр `\d{5}` и необязательно за ними следует тире и четыре цифры `-\\d{4}`. Если вы не знакомы с регулярными выражениями, возможно, сейчас самое подходящее время узнать о них ([Stroustrup,2009], [Maddock,2009], [Friedl,1997]).

Чтобы записать шаблон, я использую *сырой строковый литерал*, начинающийся с `R` и заканчивающийся на `"`. Это позволяет использовать обратную косую черту и кавычки непосредственно в строке. Сырые строки особенно подходят для регулярных выражений, поскольку они, как правило, содержат много обратных косых черт. Если бы я использовал обычную строку, определение шаблона было бы следующим:

```
regex pat {"\\w{2}\\s*\\d{5}(-\\d{4})?"};    // U.S. postal code pattern
```

В `<regex>` стандартная библиотека обеспечивает поддержку регулярных выражений:

- `regex_match()`: Сопоставление регулярного выражения со строкой (известного размера) (§10.4.2).
- `regex_search()`: Поиск строки, соответствующей регулярному выражению, в (произвольно длинном) потоке данных (§10.4.1).
- `regex_replace()`: Поиск и замена строк, соответствующих регулярному выражению, в (произвольно длинном) потоке данных.
- `regex_iterator`: Перебор совпадений и вложенных совпадений (§10.4.3).
- `regex_token_iterator`: Выполняет итерацию по несоответствиям.

10.4.1 Поиск

Самый простой способ использования шаблона - это поиск соответствия в потоке:

```
int lineno = 0;
for (string line; getline(cin,line); ) {           // read into line buffer
    ++lineno;
    smatch matches;                                // matched strings go here
    if (regex_search(line,matches,pat))             // search for pat in line
        cout << lineno << ": " << matches[0] << '\n';
}
```

`regex_search(line, matches, pat)` выполняет поиск в строке `line` всего, что соответствует регулярному выражению, хранящемуся в `pat`, и если он находит какие-либо совпадения, он сохраняет их в `matches`. Если совпадений не найдено, `regex_search(line, matches, pat)` возвращает значение `false`. Переменная `matches` имеет тип `smatch`. Буква "s" означает "sub" или "string", а `smatch` - это `vector` найденных совпадений типа `string`. Первый элемент, который здесь `matches[0]`, является полным совпадением. Результатом `regex_search()` является набор совпадений, обычно представляемый в виде `smatch`:

```
void use()
{
    ifstream in("file.txt");                       // input file
    if (!in) {                                       // check that the file was opened
        cerr << "no file\n";
        return;
    }

    regex pat {R"(\w{2}\s*\d{5}(-\d{4})?)"};        // U.S. postal code pattern

    int lineno = 0;
    for (string line; getline(in,line); ) {
        ++lineno;
```

```

    smatch matches;                                // matched strings go
here
    if (regex_search(line, matches, pat)) {
        cout << lineno << ": " << matches[0] << '\n';    // the complete match
        if (1<matches.size() && matches[1].matched)        // if there is a sub-pat-
tern
            cout << "\t: " << matches[1] << '\n';        // and if it is matched
            // submatch
    }
}

```

Эта функция читает файл в поисках почтовых индексов США, таких как **TX77845** и **DC 20500-0001**. Тип **smatch** это контейнер результатов регулярных выражений. Здесь **matches[0]** - это весь шаблон, а **matches[1]** - необязательный четырехзначный подшаблон **(-ld{4})?**.

Символ новой строки, **\n**, может быть частью шаблона, поэтому мы можем выполнять поиск многострочных шаблонов. Очевидно, если хотим это сделать, то мы не должны читать строки по одной.

Синтаксис и семантика регулярных выражений разработаны таким образом, что регулярные выражения могут быть скомпилированы в конечные автоматы для эффективного выполнения [Cox,2007]. Тип **regex** выполняет эту компиляцию во время выполнения.

10.4.2 Описание регулярных выражений

Библиотека регулярных выражений **regex** может распознавать несколько вариантов обозначения регулярных выражений. Здесь я использую обозначение по умолчанию, вариант стандарта ECMA, используемый для ECMAScript (более известного как JavaScript). Синтаксис регулярных выражений основан на *специальных символах*:

Специальные символы регулярных выражений	
.	Любой отдельный символ ("подстановочный знак")
[Начало класса символов
]	Конец класса символов
{	Начало количества вхождений
}	Конец количества вхождений
(Начало группы, подмаски
)	Конец группы, подмаски
\	Экранирование спец символа (следующий символ имеет особое значение)
*	Ни одного или более (suffix operation)
+	Одно или более (suffix operation)
?	Опционально (одно или ни одного) (suffix operation)
	Альтернатива (или)
^	Начало строки; отрицание
\$	Конец строки

Например, мы можем указать строку, начинающуюся с нуля или более **A**, за которой следует один или несколько **B**, за которыми следует необязательный **C**, как это:

^A*B+C?\$

Примеры строк, соответствующие этому выражению:

AAAAAAAAAAABBBBBBBBVC
BC
B

Примеры строк, не соответствующие выражению:

```
AAAAA           // no B
AAAABC          // initial space
AABBCC          // too many Cs
```

Часть шаблона считается подшаблоном (который может быть извлечен отдельно из `smatch`), если она заключена в круглые скобки. Например:

```
\d+-\d+          // no subpatterns
\d+(-\d+)         // one subpattern
(\d+)(-\d+)       // two subpatterns
```

Шаблон может быть необязательным или повторяющимся (по умолчанию он повторяется ровно один раз) путем добавления суффикса:

Повторение	
<code>{ n }</code>	Ровно n раз
<code>{ n, } n</code>	Или более раз
<code>{ n,m }</code>	Не менее n и не более m раз
<code>*</code>	Ни одного или более, то есть <code>{0,}</code>
<code>+</code>	Один или более, то есть <code>{1,}</code>
<code>?</code>	Необязательно (ни одного или один), то есть <code>{0,1}</code>

Например:

```
A{3}B{2,4}C*
```

Примеры соответствующих строк:

```
AAABBC
AAABBB
```

Примеры несоответствующих строк:

```
AABBC           // too few As
AAABC           // too few Bs
AAABBBBCCS     // too many Bs
```

Суффикс `?` после любого из обозначений повторения (`?`, `*`, `+` и `{ }`) делает сопоставитель шаблонов “ленивым” или “нежадным”. То есть при поиске шаблона он будет искать самое короткое совпадение, а не самое длинное. По умолчанию средство сопоставления шаблонов всегда ищет самое длинное совпадение; это известно как *правило Макса Мунка*. Рассмотрим:

```
ababab
```

Шаблону `(ab)+` соответствует весь `ababab`. Однако, `(ab)+?` соответствует только первое `ab`.

Наиболее распространенные классы символов имеют имена:

Классы символов	
<code>alnum</code>	Любой буквенно-цифровой символ
<code>alpha</code>	Любой буквенный символ
<code>blank</code>	Любой пробельный символ, который не является разделителем строк

cntrl	Любой управляющий символ
d	Любая десятичная цифра
digit	Любая десятичная цифра
graph	Любой графический символ
lower	Любой символ в нижнем регистре
print	Любой печатаемый символ
punct	Любой знак препинания
s	Любой пробельный символ
space	Любой пробельный символ
upper	Любой символ в верхнем регистре
w	Любой символ слова (буквенно-цифровые символы плюс подчеркивание)
xdigit	Любой шестнадцатеричный цифровой символ

В регулярном выражении имя символьного класса должно быть заключено в квадратные скобки `[:]`. Например, `[digit]` соответствует десятичной цифре. Кроме того, они должны использоваться в паре `[]`, определяющей класс символов.

Для некоторых классов символов поддерживается сокращенная запись:

Сокращения классов символов	
<code>\d</code> Десятичная цифра	<code>[[:digit:]]</code>
<code>\s</code> Пробел (пробел, табуляция и т.д.)	<code>[[:space:]]</code>
<code>\w</code> Буква (a-z) или цифра (0-9) или знак подчеркивания (_)	<code>[[:alnum:]]</code>
<code>\D</code> Не <code>\d</code>	<code>[^[:digit:]]</code>
<code>\S</code> Не <code>\s</code>	<code>[^[:space:]]</code>
<code>\W</code> Не <code>\w</code>	<code>[^[:alnum:]]</code>

Кроме того, языки, поддерживающие регулярные выражения, часто предоставляют:

Нестандартные (но распространенные) Сокращения классов символов	
<code>\l</code> Символ в нижнем регистре	<code>[[:lower:]]</code>
<code>\u</code> Символ в верхнем регистре	<code>[[:upper:]]</code>
<code>\L</code> Не <code>\l</code>	<code>[^[:lower:]]</code>
<code>\U</code> Не <code>\u</code>	<code>[^[:upper:]]</code>

Для обеспечения переносимости, используйте полные имена классов символов, а не сокращенные.

В качестве примера рассмотрим написание шаблона, описывающего идентификаторы C++: символ подчеркивания или буква, за которой следует, возможно, пустая последовательность букв, цифр или подчеркиваний. Чтобы проиллюстрировать связанные с этим тонкости, я приведу несколько ложных попыток:

```
[:alpha:][:alnum:]*      // wrong: characters from the set ":alpha" followed by ...
[[:alpha:]]([[:alnum:]]*) // wrong: doesn't accept underscore ( _ is not alpha)
([[:alpha:]]|_)([[:alnum:]]*) // wrong: underscore is not part of alnum either
([[:alpha:]]|_)([[:alnum:]]|_)* // OK, but clumsy
[[:alpha:]]_|[[:alnum:]]_* // OK: include the underscore in the character classes
```

```
[_[:alpha:]][_[:alnum:]]*           // also OK
[_[:alpha:]]\w*                    // \w is equivalent to [_[:alnum:]]
```

Наконец, вот функция, которая использует простейшую версию `regex_match()` (§10.4.1) для проверки того, является ли строка идентификатором:

```
bool is_identifier(const string& s)
{
    regex pat {"[_[:alpha:]]\\w*"}; // underscore or letter followed by zero or more
                                   // underscores, letters, or digits
    return regex_match(s,pat);
}
```

Обратите внимание на удвоение обратной косой черты, чтобы включить обратную косую черту в обычный строковый литерал. Используйте необработанные строковые литералы (§10.4), чтобы устранить проблемы со специальными символами. Например:

```
bool is_identifier(const string& s)
{
    regex pat {R"([_[:alpha:]]\\w*)"};
    return regex_match(s,pat);
}
```

Вот несколько примеров шаблонов::

```
Ax*           // A, Ax, Axxxx
Ax+           // Ax, Axxx           Not A
\d-?\d        // 1-2, 12           Not 1--2
\w{2}-\d{4,5} // Ab-1234, XX-54321, 22-5432   Digits are in \w
(\d*:\d+)?    // 12:3, 1:23, 123, :123       Not 123:
(bs|BS)       // bs, BS             Not bS
[aeiouy]      // a, o, u            An English vowel, not x
[^aeiouy]     // x, k              Not an English vowel, not e
[a^aeiouy]    // a, ^, o, u        An English vowel or ^
```

group (подшаблон), который потенциально может быть представлен с помощью `sub_match`, взят в круглые скобки. Если вам нужны круглые скобки, которые не должны определять подшаблон, используйте `(?:` вместо простой `(`. Например:

```
(\s|:|,)*(\\d*) // optional spaces, colons, and/or commas followed by an optional
number
```

Предполагая, что нас не интересовали символы перед числом (предположительно разделители), мы могли бы написать:

```
(?:\s|:|,)*(\\d*) // optional spaces, colons, and/or commas followed by an op-
tional number
```

Это избавило бы механизм регулярных выражений от необходимости сохранять первые символы: вариант `(?:` имеет только один подшаблон.

Примеры группировки регулярных выражений

<code>\d*\s\w+</code>	Нет групп (подшаблонов)
<code>(\d*)\s(\w+)</code>	Две группы
<code>(\d*)(\s(\w+))+</code>	Две группы (группы не объединяются)
<code>(\s*\w*)+</code>	Одна группа; один или несколько вложенных шаблонов; только последний подшаблон сохраняется как <code>sub_match</code>

`<(.*?)>(.*?)` Три группы; `\1` означает “то же, что и в группе 1”.

Этот последний шаблон полезен для синтаксического анализа XML. Он находит маркеры начала/конца тега. Обратите внимание, что я использовал нежадное совпадение (*ленивое совпадение*), `.*?`, для подшаблона между тегом и конечным тегом. Если бы я использовал обычный `.*`, этот ввод вызвал бы проблему:

Always look on the **bright** side of **life**.

Жадный поиск для первого подшаблона соответствовало бы первому `<` с последним `>`. Это было бы правильным поведением, но вряд ли то, чего хотел программист.

Для более исчерпывающего представления регулярных выражений смотрите [\[Friedl,1997\]](#).

10.4.3 Итераторы

Мы можем определить `regex_iterator` для перебора последовательности совпадений с шаблоном. Например, мы можем использовать `sregex_iterator` (`regex_iterator<string>`) для вывода всех слов, разделенных пробелами, в `string`:

```
void test()
{
    string input = "aa as; asd ++e^asdf asdfg";
    regex pat {R"(\s+(\w+))"};
    for (sregex_iterator p(input.begin(),input.end(),pat); p!=sregex_iterator{}; ++p)
        cout << (*p)[1] << '\n';
}
```

Результат работы кода:

```
as
asd
asdfg
```

Мы пропустили первое слово, `aa`, потому что перед ним нет пробелов. Если мы упростим шаблон до `R"((\w+))"`, то получим

```
aa
as
asd
e
asdf
asdfg
```

`regex_iterator` - это двунаправленный итератор, поэтому мы не можем напрямую выполнять итерацию по `istream` (который предлагает только входной итератор). Кроме того, мы не можем выполнять запись через `regex_iterator`, и `regex_iterator` по умолчанию (`regex_iterator{}`) является единственно возможным концом последовательности.

10.5 Советы

- [1] Используйте `std::string` для создания собственных последовательностей символов; §10.2; [CG: SL.str.1].
- [2] Предпочитайте операции со `string` функциям строк в стиле C; §10.1.
- [3] Используйте `string` для объявления переменных и членов, а не в качестве базового класса; §10.2.

- [4] Возвращайте `string` по значению (полагаясь на семантику перемещения и исключение копирования); §10.2, §10.2.1; [CG: F.15].
- [5] Прямо или косвенно используйте `substr()` для чтения подстрок и `replace()` для записи подстрок; §10.2.
- [6] `string` может увеличиваться и уменьшаться по мере необходимости; §10.2.
- [7] Используйте `at()` вместо итераторов или `[]`, когда вы хотите проверить диапазон; §10.2, §10.3.
- [8] Используйте итераторы и `[]` вместо `at()`, когда вы хотите оптимизировать скорость; §10.2, §10.3.
- [9] Используйте диапазонный `for`, чтобы безопасно свести к минимуму проверку диапазона §10.2, §10.3.
- [10] Запись в `string` не приводит к переполнению; §10.2, §11.3.
- [11] Используйте `c_str()` или `data()` для создания представления строки в стиле C из `string` (только) когда это необходимо; §10.2.
- [12] Используйте `stringstream` или универсальную функцию извлечения значений (например, `to<X>`) для числового преобразования строк; §11.7.3.
- [13] `basic_string` можно использовать для создания строк символов любого типа; §10.2.1.
- [14] Используйте суффикс `s` для строковых литералов, которые должны быть строками `string` стандартной библиотеки; §10.3 [CG: SL.str.12].
- [15] Используйте `string_view` в качестве аргумента функций, которым необходимо считывать последовательности символов, хранящихся различными способами; §10.3 [CG: SL.str.2].
- [16] Используйте `string_span<char>` в качестве аргумента функций, которым необходимо записать последовательности символов, хранящихся различными способами; §10.3. [CG: SL.str.2] [CG: SL.str.11].
- [17] Думайте о `string_view` как о своего рода указателе с привязанным размером; он не владеет своими символами; §10.3.
- [18] Используйте суффикс `sv` для строковых литералов, которые должны быть типа `string_view` стандартной библиотеки; §10.3.
- [19] Используйте `regex` для большинства обычных применений регулярных выражений; §10.4.
- [20] Предпочитайте необработанные строковые литералы для выражения всех шаблонов, кроме простейших; §10.4.
- [21] Используйте `regex_match()` для поиска в законченном вводе; §10.4, §10.4.2.
- [22] Используйте `regex_search()` для поиска шаблона в потоке ввода; §10.4.1.
- [23] Обозначение регулярных выражений может быть скорректировано в соответствии с различными стандартами; §10.4.2.
- [24] Обозначение регулярных выражений по умолчанию взято из ECMAScript; §10.4.2.
- [25] Будьте сдержанны; регулярные выражения могут легко стать нечитаемыми; §10.4.2.
- [26] Обратите внимание, что `\i` для цифры `i` позволяет вам выразить подшаблон в терминах предыдущего подшаблона; §10.4.2.
- [27] Используйте `?` чтобы сделать шаблоны “ленивыми”; §10.4.2.
- [28] Используйте `regex_iterator` для перебора потока в поисках шаблона; §10.4.3.

Ввод и вывод

*Что видите, то и получите.
– Brian W. Kernighan*

- [Введение](#)
- [Вывод](#)
- [Ввод](#)
- [Состояния потоков I/O](#)
- [I/O пользовательских типов](#)
- [Форматирование](#)

[Форматирование потока](#); [printf\(\)-style форматирование](#)

- [Потоки](#)

[Стандартные потоки](#); [Файловые потоки](#); [Строковые потоки](#); [Потоки памяти](#); [Синхронные потоки](#)

- [I/O в стиле Си](#)
- [Файловая система](#)

[Пути](#); [Файлы и каталоги](#)

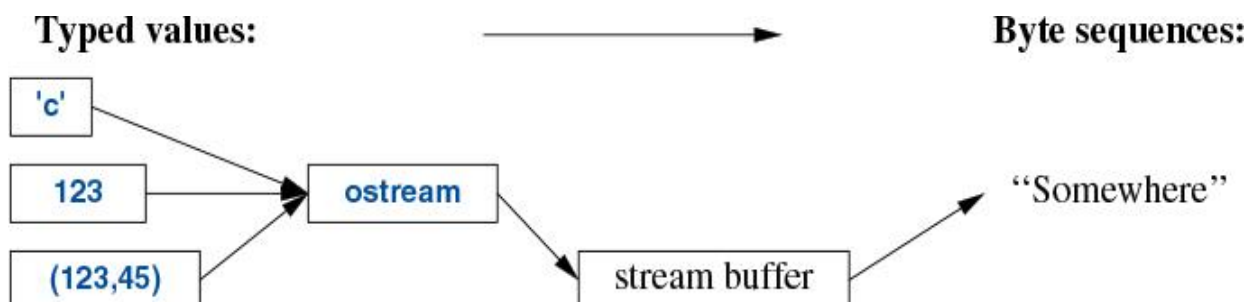
- [Советы](#)

11.1 Введение

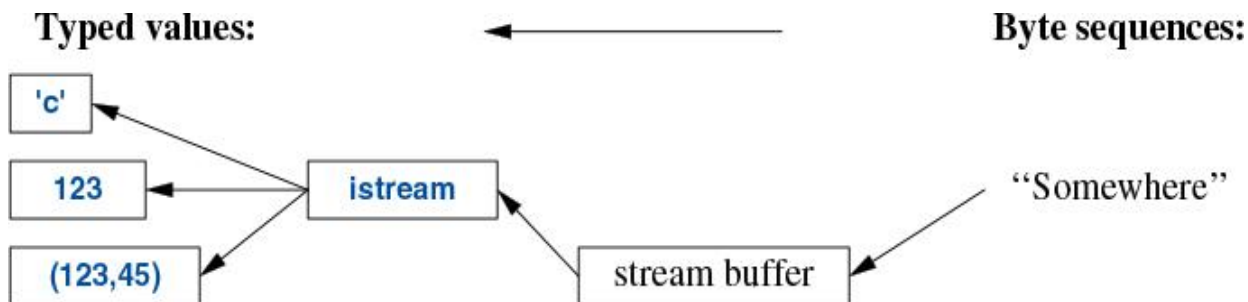
Библиотека потоков ввода-вывода обеспечивает форматированный и неформатированный буферизованный ввод-вывод текстовых и числовых значений. Он расширяем для поддержки пользовательских типов точно так же, как встроенных типов и является типобезопасным.

Библиотека файловой системы предоставляет базовые средства для работы с файлами и каталогами.

`ostream` преобразует типизированные объекты в поток символов (байтов):



`istream` преобразует поток символов (байтов) в типизированные объекты:



Операции с `istream` и `ostream` описаны в §11.2 и §11.3. Операции являются типобезопасными, чувствительными к типу и расширяемыми для обработки пользовательских типов (§11.5).

Другие формы взаимодействия с пользователем, такие как графический ввод-вывод, обрабатываются с помощью библиотек, которые не являются частью стандарта ISO и, следовательно, здесь не описаны.

Рассматриваемые потоки могут использоваться для двоичного ввода-вывода, использоваться для различных типов символов, зависеть от локали и использовать расширенные стратегии буферизации, но эти темы выходят за рамки данной книги.

Потоки могут использоваться для ввода в `string` и вывода из них (§11.3), для форматирования в `string` буферы (§11.7.3), в области памяти (§11.7.4) и для файлового ввода-вывода (§11.9).

Все классы потоков ввода-вывода имеют деструкторы, которые освобождают все принадлежащие ресурсы (такие как буферы и файловые дескрипторы). То есть, они являются примерами применения идиомы "Получение ресурса - это инициализация" (RAII; §6.3).

11.2 Вывод

В `<ostream>` библиотеке потоков ввода-вывода определён вывод для каждого встроенного типа. Кроме того, легко определить вывод пользовательского типа (§11.5). Оператор `<<` ("put to") используется в качестве оператора вывода в объекты типа `ostream`; `cout` - это стандартный поток вывода, а `cerr` - стандартный поток для сообщений об ошибках. По умолчанию значения, записанные в `cout`, преобразуются в последовательность символов. Например, чтобы вывести десятичное число `10`, мы можем записать:

```
cout << 10;
```

Это помещает символ `1`, за которым следует символ `0`, в стандартный поток вывода. Эквивалентно, мы могли бы написать:

```
int x {10};
cout << x;
```

Выходные данные различных типов могут быть объединены очевидным образом:

```
void h(int i)
{
    cout << "the value of i is ";
    cout << i;
    cout << '\n';
}
```

Для `h(10)` будет выведено:

```
the value of i is 10
```

Программисты быстро устают повторять название выходного потока при выводе нескольких связанных элементов. К счастью, результат выходного выражения сам по себе может быть использован для дальнейшего вывода. Например:

```
void h2(int i)
{
    cout << "the value of i is " << i << '\n';
}
```

Этот `h2()` выдает тот же результат, что и `h()`.

Символьная константа - это символ, заключенный в одинарные кавычки. Обратите внимание, что символ выводится как символ, а не как числовое значение. Например:

```
int b = 'b';           // note: char implicitly converted to int
char c = 'c';
cout << 'a' << b << c;
```

Целочисленное значение символа `'b'` равно **98** (в кодировке ASCII, используемой в реализации C++), так что это выведет **a98c**.

11.3 Ввод

В `<istream>` стандартная библиотека предлагает `istream` потоки для ввода. Как и `ostream`, `istream` работают с символьными строковыми представлениями встроенных типов и могут быть легко расширены для работы с пользовательскими типами.

Оператор `>>` ("get from") используется в качестве оператора ввода; `cin` - это стандартный поток ввода. Тип правого операнда `>>` определяет, какие входные данные принимаются и какова цель операции ввода. Например:

```
int i;
cin >> i;           // read an integer into i

double d;
cin >> d;           // read a double-precision floating-point number into d
```

В первом случае считывает число, например **1234**, из стандартного ввода в целочисленную переменную `i`, во втором - число с плавающей точкой, например **12.34e5**, в переменную с плавающей точкой двойной точности `d`.

Как и операции вывода, операции ввода могут быть связаны цепочкой, так что я мог бы эквивалентно написать:

```
int i;
double d;
cin >> i >> d;      // read into i and d
```

В обоих случаях чтение целого числа завершается любым символом, который не является цифрой. По умолчанию `>>` пропускает начальные пробельные символы, поэтому подходящей полной последовательностью ввода будет

```
1234
12.34e5
```

Часто мы хотим прочитать последовательность символов. Удобный способ сделать это - прочитать в `string`. Например:

```
cout << "Please enter your name\n";
string str;
```

```
cin >> str;
cout << "Hello, " << str << "!\n";
```

Если вы введете **Eric**, ответ будет таким:

```
Hello, Eric!
```

По умолчанию чтение завершается пробельным символом, таким как пробел или новая строка, поэтому, если вы введете **Eric Bloodaxe** ответ по-прежнему будет:

```
Hello, Eric!
```

Вы можете прочитать целую строку, используя функцию **getline()**. Например:

```
cout << "Please enter your name\n";
string str;
getline(cin, str);
cout << "Hello, " << str << "!\n";
```

С помощью этой программы ввод **Eric Bloodaxe** приводит к желаемому результату:

```
Hello, Eric Bloodaxe!
```

Перевод строки, завершавший вводимую строку, отбрасывается, поэтому **cin** готов к вводу следующей строки.

Использование форматированных операций ввода-вывода обычно менее подвержено ошибкам, более эффективно и требует меньше кода, чем манипулирование символами по одному. В частности, **istream** заботится об управлении памятью и проверке диапазона. Мы можем выполнять форматирование в памяти и вывод из нее, используя **stringstream** (§11.7.3) или **memory streams** (§11.7.4).

Стандартные строки обладают приятным свойством расширяться, чтобы вместить то, что вы в них помещаете; вам не нужно предварительно вычислять максимальный размер. Итак, если вы введете пару мегабайт точек с запятой, функция **hello_line()** вернет вам страницы с точками с запятой.

11.4 Состояния потоков I/O

У **istream** есть состояние, которое мы можем проверить, чтобы определить, была ли операция выполнена успешно. Наиболее распространенным способом является считывание последовательности значений:

```
vector<int> read_ints(istream& is)
{
    vector<int> res;
    for (int i; is>>i; )
        res.push_back(i);
    return res;
}
```

Этот код считывает из **is** до тех пор, пока не встретится что-то, что не является целым числом. Это что-то, как правило, будет концом ввода. Что здесь происходит, так это то, что операция **is>>i** возвращает ссылку на **is**, а тестирование **istream** выдает значение **true**, если поток готов к следующей операции.

В общем, состояние ввода-вывода содержит всю информацию, необходимую для чтения или записи, такую как информация о форматировании (§11.6.2), состояние ошибки (например, достигнут ли конец ввода?) и какой тип буферизации используется. В частности, пользователь может настроить состояние таким образом, чтобы оно

отражало, что произошла ошибка (§11.5), и очистить состояние, если ошибка не была серьезной. Например, мы могли бы представить версию `read_ints()`, которая принимала бы строку завершающую ввод:

```
vector<int> read_ints(istream& is, const string& terminator)
{
    vector<int> res;
    for (int i; is >> i; )
        res.push_back(i);

    if (is.eof())                // fine: end of file
        return res;
    if (is.fail()) {             // we failed to read an int; was it the termina-
tor?
        is.clear();              // reset the state to good()
        string s;
        if (is>>s && s==terminator)
            return res;
        is.setstate(ios_base::failbit); // add fail() to is's state
    }
    return res;
}

auto v = read_ints(cin,"stop");
```

11.5 I/O пользовательских типов

В дополнение к вводу-выводу встроенных типов и стандартных строк `string` библиотека `iostream` позволяет нам определять ввод-вывод для наших собственных типов. Например, рассмотрим простой тип `Entry`, который мы могли бы использовать для представления записей в телефонной книге:

```
struct Entry {
    string name;
    int number;
};
```

Мы можем определить простой оператор вывода для вывода `Entry`, используя формат `{"name",number}`, аналогичный тому, который мы используем для инициализации в коде:

```
ostream& operator<<(ostream& os, const Entry& e)
{
    return os << "{" << e.name << "\", " << e.number << "}";
}
```

Пользовательский оператор вывода принимает в качестве первого аргумента выходной поток (по ссылке) и возвращает его в качестве результата.

Соответствующий оператор ввода является более сложным, поскольку он должен проверять правильность форматирования и обрабатывать ошибки:

```
istream& operator>>(istream& is, Entry& e)
    // read { "name" , number } pair. Note: formatted with { " " , and }
{
    char c, c2;
    if (is>>c && c=='{' && is>>c2 && c2== '\'"' ) { // start with a { followed by a
"
        string name; // the default value of a string is the empty string:
""
        while (is.get(c) && c!= '\'"' ) // anything before a " is part of the
```

```

name
    name+=c;

    if (is>>c && c==',') {
        int number = 0;
        if (is>>number>>c && c=='}') { // read the number and a }
            e = {name,number};        // assign to the entry
            return is;
        }
    }
}

is.setstate(ios_base::failbit); // register the failure in the stream
return is;
}

```

Операция ввода возвращает ссылку на свой **istream**, которую можно использовать для проверки успешности выполнения операции. Например, использование в качестве условия **is>>c** означает “Удалось ли нам прочитать **char** из **is** в **c**?”

По умолчанию **is>>c** пропускает пробельные символы, а **is.get(c)** этого не делает, поэтому этот оператор ввода **Entry** игнорирует (пропускает) пробелы вне строки имени, но не внутри нее. Например:

```

{ "John Marwood Cleese", 123456      }
{"Michael Edward Palin", 987654}

```

Мы можем прочитать такую пару значений из входных данных в **Entry**, например, так:

```

for (Entry ee; cin>>ee; ) // read from cin into ee
    cout << ee <<'\n';   // write ee to cout

```

Вывод в таком случае:

```

{"John Marwood Cleese", 123456}
{"Michael Edward Palin", 987654}

```

Смотрите §10.4 для более систематизированного метода распознавания шаблонов в потоках символов (сопоставление с регулярными выражениями).

11.6 Форматирование вывода

Библиотеки **iostream** и **format** предоставляют операции для управления форматом ввода и вывода. Средства **iostream** примерно такие же старые, как C++, и ориентированы на форматирование потоков чисел. Средства **format** (§11.6.2) появились недавно (C++20) и ориентированы на форматирование комбинаций значений по спецификации в стиле **printf()** (§11.8).

Форматирование выходных данных также обеспечивает поддержку **unicode**, но это выходит за рамки данной книги.

11.6.1 Форматирование потока

Простейшие элементы управления форматированием называются *манипуляторами* и находятся в **<ios>**, **<istream>**, **<ostream>** и **<iomanip>** (для манипуляторов, принимающих аргументы). Например, мы можем выводить целые числа в виде десятичных (по умолчанию), восьмеричных или шестнадцатеричных чисел:

```

cout << 1234 <<' ' << hex << 1234 <<' ' << oct << 1234 << dec << 1234 <<'\n';

```

```
// 1234 4d2 2322 1234
```

Мы можем явно задать формат вывода для чисел с плавающей запятой:

```
constexpr double d = 123.456;
```

```
cout << d << "; "           // use the default format for d
    << scientific << d << "; " // use 1.123e2 style format for d
    << hexfloat << d << "; "   // use hexadecimal notation for d
    << fixed << d << "; "     // use 123.456 style format for d
    << defaultfloat << d << '\n'; // use the default format for d
```

Соответственно вывод:

```
123.456; 1.234560e+002; 0x1.edd2f2p+6; 123.456000; 123.456
```

Точность - это целое число, определяющее количество цифр, используемых для отображения числа с плавающей запятой:

- *Общий* формат (**defaultfloat**) позволяет реализации выбрать формат, который представляет значение в таком стиле, который наилучшим образом сохраняет значение в доступном количестве знакомест. Точность определяет максимальное количество цифр.
- *Научный* формат (**scientific**) представляет значение с одной цифрой перед десятичной точкой и показателем степени. Точность определяет максимальное количество цифр после десятичной точки.
- *Фиксированный* формат (**fixed**) представляет значение в виде целой части, за которой следуют десятичная точка и дробная часть. Точность определяет максимальное количество цифр после десятичной точки.

Значения с плавающей точкой округляются, а не просто усекаются, и функция **precision()** не влияет на вывод целых чисел. Например:

```
cout.precision(8);
cout << "precision(8):" << 1234.56789 << ' ' << 1234.56789 << ' ' << 123456 << '\n';

cout.precision(4);
cout << "precision(4): " << 1234.56789 << ' ' << 1234.56789 << ' ' << 123456 << '\n';
cout << 1234.56789 << '\n';
```

Соответственно вывод:

```
precision(8): 1234.5679 1234.5679 123456
precision(4): 1235 1235 123456
1235
```

Эти манипуляторы с плавающей точкой являются “липкими”; то есть их эффекты сохраняются для последующих операций с плавающей точкой. То есть они предназначены в первую очередь для форматирования потоков значений.

Мы также можем указать размер поля, в которое должно быть помещено число, и его выравнивание в этом поле.

В дополнение к основным числам, **<<** также может обрабатывать время и даты: **duration**, **time_point** **year_month_date**, **weekday**, **month** и **zoned_time** (§16.2). Например:

```
cout << "birthday: " << November/28/2021 << '\n';
cout << << "zt: " << zoned_time{current_zone(), system_clock::now()} << '\n';
```

Соответственно вывод:

birthday: 2021-11-28
zt: 2021-12-05 11:03:13.5945638 EST

Стандарт также определяет `<<` для `complex`, `bitset` (§15.3.2), кодов ошибок и указателей. Поточковый ввод-вывод является расширяемым, поэтому мы можем определить `<<` для наших собственных (определяемых пользователем) типов (§11.5).

11.6.2 Форматирование в стиле `printf()`

Было убедительно доказано, что `printf()` является самой популярной функцией в C и важным фактором ее успеха. Например:

```
printf("an int %g and a string '%s'\n",123,"Hello!");
```

Этот стиль “форматирования строки, за которой следуют аргументы” был заимствован в C из BCPL и используется многими языками. Естественно, `printf()` всегда была частью стандартной библиотеки C++, но она страдает от отсутствия безопасности типов и расширяемости для обработки пользовательских типов.

В `<format>` стандартная библиотека предоставляет типобезопасный, хотя и не расширяемый механизм форматирования в стиле `printf()`. Базовая функция `format()` выдает `string`:

```
string s = format("Hello, {}\n", val);
```

“Обычные символы” в *форматируемой строке* просто вставляются в выходной `string`. *Форматируемая строка*, разделенная символами `{}` и `}`, указывает, как аргументы, следующие за строкой формата, должны быть вставлены в `string`. Самая простая строка формата - это пустая строка `{}`, которая принимает следующий аргумент из списка аргументов и печатает его в соответствии с его оператором `<<` по умолчанию (если таковое имеется). Итак, если `val` - это `"World"`, мы получаем культовое `"Hello, World\n"`. Если `val` равно `127`, мы получаем `"Hello, 127\n"`.

Наиболее распространенным использованием `format()` является вывод его результата:

```
cout << format("Hello, {}\n", val);
```

Чтобы увидеть, как это работает, давайте сначала повторим примеры из (§11.6.1):

```
cout << format("{} {:x} {:o} {:d} {:b}\n", 1234,1234,1234,1234,1234);
```

Это дает тот же результат, что и в примере с целым числом в §11.6.1, за исключением того, что я добавил `b` для двоичного файла, который напрямую не поддерживается `ostream`:

```
1234 4d2 2322 1234 10011010010
```

Перед директивой форматирования ставится двоеточие. Альтернативными вариантами целочисленного форматирования являются `x` для шестнадцатеричного, `o` для восьмеричного, `d` для десятичного и `b` для двоичного.

По умолчанию `format()` принимает свои аргументы по порядку. Однако мы можем указать произвольный порядок. Например:

```
cout << format("{3:} {1:x} {2:o} {0:b}\n", 000, 111, 222, 333);
```

Это выводит `333 6f 336 0`. Число перед двоеточием - это номер форматируемого аргумента. В лучших традициях C++ нумерация начинается с нуля. Данная запись позволяет нам форматировать аргумент более одного раза:

```
cout << format("{0:} {0:x} {0:o} {0:d} {0:b}\n", 1234); // default, hexadecimal, oc-
tal, decimal, binary
```

Возможность помещать аргументы в выходные данные “не по порядку” высоко ценится людьми, создающими сообщения на разных естественных языках.

Форматы с плавающей запятой такие же, как и для **ostream**: **e** для scientific, **a** для hexfloat, **f** для fixed и **g** для default. Например:

```
cout << format("{0:}; {0:e}; {0:a}; {0:f}; {0:g}\n",123.456); // default, scien-
tific, hexfloat, fixed, default
```

Результат идентичен результату из **ostream**, за исключением того, что шестнадцатеричному числу не предшествовал **0x**:

```
123.456; 1.234560e+002; 1.edd2f2p+6; 123.456000; 123.456
```

Точка предшествует указателю точности:

```
cout << format("precision(8): {:.8} {} {} \n", 1234.56789, 1234.56789, 123456);
cout << format("precision(4): {:.4} {} {} \n", 1234.56789, 1234.56789, 123456);
cout << format("{} \n", 1234.56789);
```

В отличие от потоков, спецификаторы не являются “липкими”, поэтому мы получаем:

```
precision(8): 1234.5679 1234.56789 123456
precision(4): 1235 1234.56789 123456
1234.56789
```

Как и в случае с потоковыми форматировщиками, мы также можем указать размер поля, в которое должно быть помещено число, и его выравнивание в этом поле.

Подобно форматировщикам потоков, **format()** также может обрабатывать время и даты (§16.2.2). Например:

```
cout << format("birthday: {} \n",November/28/2021);
cout << format("zt: {}", zoned_time{current_zone(), system_clock::now()});
```

Как обычно, форматирование значений по умолчанию идентично форматированию вывода потока по умолчанию. Однако **format()** предлагает мини-язык, содержащий около 60 спецификаторов формата, позволяющий очень детально управлять форматированием чисел и дат. Например:

```
auto ymd = 2021y/March/30 ;
cout << format("ymd: {3:%A},{1:} {2:%B},{0:} \n", ymd.year(), ymd.month(), ymd.day(),
weekday(ymd));
```

Результат вывода:

```
ymd: Tuesday, March 30, 2021
```

Все строки формата времени и даты начинаются с %.

Гибкость, обеспечиваемая многими спецификаторами формата, может быть важна, но она сопряжена со многими возможностями для ошибок. Некоторые спецификаторы поставляются с необязательной семантикой или семантикой, зависящей от языка. Если ошибка форматирования обнаруживается во время выполнения, генерируется исключение **format_error**. Например:

```
string ss = format("{:%F}", 2); // error: mismatched argument; potentially caught at
// compile time
```

```
string sss = format("{%F}", 2); // error: bad format; potentially caught at compile
time
```

Примеры до сих пор имели постоянные форматы, которые можно проверить во время компиляции. Дополнительная функция `vformat()` использует переменную в качестве формата для значительного повышения гибкости и возможности возникновения ошибок во время выполнения:

```
string fmt = "{}";
cout << vformat(fmt, make_format_args(2)); // OK
fmt = "{:%F}";
cout << vformat(fmt, make_format_args(2)); // error: format and argument mismatch;
caught at run time
```

Наконец, средство форматирования также может выполнять запись непосредственно в буфер, определенный итератором. Например:

```
string buf;
format_to(back_inserter(buf), "iterator: {} {}\\n", "Hi! ", 2022);
cout << buf; // iterator: Hi! 2022
```

Это становится интересным с точки зрения производительности, если мы используем буфер потока напрямую или буфер для какого-либо другого устройства вывода.

11.7 Потоки

Стандартная библиотека непосредственно поддерживает

- *Стандартные потоки*: потоки, подключенные к стандартным потокам ввода-вывода системы (§11.7.1)
- *Файловые потоки*: потоки, прикрепленные к файлам (§11.7.2)
- *Строковые потоки*: потоки, присоединенные к строкам (§11.7.3)
- *Потоки памяти*: поток, привязанный к определенным областям памяти (§11.7.4)
- *Синхронизированные потоки*: потоки, которые могут использоваться из нескольких потоков без скачков данных (§11.7.5)

Кроме того, мы можем определить наши собственные потоки, например, подключенные к каналам связи.

Потоки не могут быть скопированы; всегда передавайте их по ссылке.

Все потоки стандартной библиотеки являются шаблонами с типом символа в качестве параметра. Версии с названиями, которые я здесь использую, содержат `char`. Например, `ostream` - это `basic_ostream<char>`. Для каждого такого потока стандартная библиотека также предоставляет версию для `wchar_t`. Например, `wostream` - это `basic_ostream<wchar_t>`. Потоки широких символов можно использовать для символов Юникода.

11.7.1 Стандартные потоки

Стандартными потоками являются

- `cout` для “обычного вывода”
- `cerr` для небуферизованного “вывода ошибки”
- `clog` буферизованного “вывода журнала”
- `cin` для стандартного ввода.

11.7.2 Файловые потоки

В `<fstream>` стандартной библиотеки представлены потоки чтения и записи в файл:

- `ifstream` для чтения из файла
- `ofstream` для записи в файл
- `fstream` для чтения из файла и записи в него, например:

```
ofstream ofs {"target"};                // "o" for "output"
if (!ofs)
    error("couldn't open 'target' for writing");
```

Проверка правильности открытия файлового потока обычно выполняется путем проверки его состояния.

```
ifstream ifs {"source"};                // "i" for "input"
if (!ifs)
    error("couldn't open 'source' for reading");
```

Предполагая, что тесты прошли успешно, `ofs` можно использовать как обычный `ostream` (точно так же, как `cout`), а `ifs` можно использовать как обычный `istream` (точно так же, как `cin`).

Навигация в файле и более детальное управление способом открытия файла возможны, но выходят за рамки данной книги.

О составлении имен файлов и манипуляциях с файловой системой смотрите в §11.9.

11.7.3 Строковые потоки

В `<sstream>`, стандартная библиотека предоставляет потоки для чтения и записи в строку:

- `istringstream` для чтения из `string`
- `ostringstream` для записи в `string`
- `stringstream` для чтения и записи в `string`.

Например:

```
void test()
{
    ostringstream oss;

    oss << "{temperature," << scientific << 123.4567890 << "}";
    cout << oss.view() << '\n';
}
```

Содержимое `ostringstream` может быть прочитано с помощью `str()` (содержимое в виде копии `string`) или `view()` (содержимое в виде `string_view`). Одним из распространенных способов использования `ostringstream` является форматирование перед передачей результирующей строки в графический интерфейс. Аналогично, строку, полученную из графического интерфейса пользователя, можно прочитать с помощью операций форматированного ввода (§11.3), поместив ее в `istringstream`.

`stringstream` можно использовать как для чтения, так и для записи. Например, мы можем определить операцию, которая может преобразовать любой тип представленный в виде `string` в другой, который также может быть представлен в виде `string`:

```
template<typename Target =string, typename Source =string>
Target to(Source arg)                // convert Source to Target
{
```



```

stringstream buf;
Target result;

if (!(buf << arg)                // write arg into stream
    || !(buf >> result)           // read result from stream
    || !(buf >> std::ws).eof())    // is anything left in stream?
    throw runtime_error{"to<>() failed"};

return result;
}

```

Аргумент шаблона функции должен быть явно указан только в том случае, если он не может быть выведен или если значение по умолчанию отсутствует (§8.2.4), поэтому мы можем написать:

```

auto x1 = to<string,double>(1.2); // very explicit (and verbose)
auto x2 = to<string>(1.2);         // Source is deduced to double
auto x3 = to<>(1.2);               // Target is defaulted to string; Source is de-
    deduced                        // to double
auto x4 = to(1.2);                 // the <> is redundant;
    deduced                        // Target is defaulted to string; Source is de-
    deduced                        // to double

```

Если все аргументы шаблона функции заданы по умолчанию, угловые скобки <> можно опустить.

Я считаю это хорошим примером универсальности и простоты использования, которые могут быть достигнуты сочетанием возможностей языка и стандартной библиотеки.

11.7.4 Потоки памяти

С самых ранних дней C++ существовали потоки, привязанные к разделам памяти, указанным пользователем, так что мы могли читать/записывать непосредственно в них. Самые старые из таких потоков, **stringstream**, устарели на протяжении десятилетий, но их замена, **spanstream**, **ispanstream** и **ospanstream**, станет официальной не ранее C++23. Однако они уже широко доступны; попробуйте свою реализацию или выполните поиск на GitHub.

ospanstream ведет себя как **ostream** (§11.7.3) и инициализируется аналогично ему, за исключением того, что **ospanstream** принимает в качестве аргумента **span**, а не **string**. Например:

```

void user(int arg)
{
    array<char,128> buf;
    ospanstream ss(buf);
    ss << "write " << arg << " to memory\n";
    // ...
}

```

Попытки переполнения целевого буфера приводят к **failure** (сбою) состояния строки (§11.4).

Аналогично, **ispanstream** подобен **istream**.

11.7.5 Синхронизированные потоки

В многопоточной системе ввод-вывод становится ненадежным и беспорядочным, если не:

- Только один **thread** использует **stream**.
- Доступ к **stream** синхронизируется таким образом, что одновременно доступ получает только один **thread**.

osyncstream гарантирует, что последовательность операций вывода будет завершена и их результаты будут такими, как ожидалось, в выходном буфере, даже если какой-либо другой **thread** попытается выполнить запись. Например:

```
void unsafe(int x, string& s)
{
    cout << x;
    cout << s;
}
```

Другой **thread** может вызвать состояние гонки (§18.2) и привести к неожиданному результату. Чтобы избежать этого, можно использовать **osyncstream**

```
void safer(int x, string& s)
{
    osyncstream oss(cout);
    oss << x;
    oss << s;
}
```

Другие **thread**, которые также используют **osyncstream**, не будут вмешиваться. Другой **thread**, который напрямую использует **cout**, может создавать помехи, поэтому либо используйте **ostrstream** повсеместно, либо убедитесь, что только один **thread** выдает выходные данные в определенный выходной поток.

Параллелизм может быть сложным, поэтому будьте осторожны (глава 18). Избегайте совместного использования данных между **thread**ами, когда это возможно.

11.8 I/O в стиле Си

Стандартная библиотека C++ также поддерживает ввод-вывод стандартной библиотеки C, включая **printf()** и **scanf()**. Многие варианты использования этой библиотеки ненадежны с точки зрения типовой и кибер-безопасности, поэтому я не рекомендую ее использовать. В частности, её может быть сложно использовать для безопасного и удобного ввода. Она не поддерживает пользовательские типы. Если вы не используете ввод-вывод в стиле C, и заботитесь о производительности ввода-вывода, вызовите

```
ios_base::sync_with_stdio(false); // avoid significant overhead
```

Без этого вызова стандартные потоки **iostream** (например, **cin** и **cout**) могут быть значительно замедлены, чтобы быть совместимыми с вводом-выводом в стиле C.

Если вам нравится форматированный вывод в стиле **printf()**, используйте **format** (§11.6.2); он безопасен для ввода, прост в использовании, гибок и быстр.

11.9 Файловая система

Большинство систем имеют описание *файловой системы*, обеспечивающее доступ к постоянной информации, хранящейся в виде *файлов*. К сожалению, свойства файловых систем и способы манипулирования ими сильно различаются. Чтобы справиться с этим, библиотека файловой системы в **<filesystem>** предлагает единый интерфейс для

большинства средств большинства файловых систем. Используя `<filesystem>`, мы можем переносимо

- определять пути в файловой системе и перемещаться по ней
- узнавать типы файлов и связанные с ними разрешения

Библиотека файловой системы может обрабатывать Unicode, но объяснение того, как это делается, выходит за рамки данной книги. Я рекомендую *cppreference* [[Cppreference](#)] и документацию по *Boost filesystem* [[Boost](#)] для получения более подробной информации.

11.9.1 Пути

Рассмотрим пример:

```
path f = "dir/hypothetical.cpp";    // naming a file

assert(exists(f));                  // f must exist

if (is_regular_file(f))             // is f an ordinary file?
    cout << f << " is a file; its size is " << file_size(f) << '\n';
```

Обратите внимание, что программа, управляющая файловой системой, обычно выполняется на компьютере вместе с другими программами. Таким образом, содержимое файловой системы может изменяться между двумя командами. Например, несмотря на то, что мы сначала тщательно утверждали, что `f` существует, это может уже не соответствовать действительности, когда в следующей строке мы спрашиваем, является ли `f` обычным файлом.

Путь `path` - это довольно сложный класс, способный обрабатывать различные наборы символов и соглашения многих операционных систем. В частности, он может обрабатывать имена файлов из командной строки, представленные `main()`; например:

```
int main(int argc, char* argv[])
{
    if (argc < 2) {
        cerr << "arguments expected\n";
        return 1;
    }

    path p {argv[1]};                // create a path from the command line

    cout << p << " " << exists(p) << '\n';    // note: a path can be printed like a
    // ...                                string
}
```

Путь `path` не проверяется на достоверность до тех пор, пока он не будет использован. Даже в этом случае его действительность зависит от соглашений системы, в которой выполняется программа.

Естественно, можно использовать `path` для открытия файла:

```
void use(path p)
{
    ofstream f {p};
    if (!f) error("bad file name: ", p);
    f << "Hello, file!";
}
```

В дополнение к `path`, `<filesystem>` предлагает типы для обхода каталогов и запроса свойств найденных файлов:

Типы для работы с файловой системой (некоторые)	
<code>path</code>	Путь к каталогу
<code>filesystem_error</code>	Исключение файловой системы
<code>directory_entry</code>	Запись в каталоге
<code>directory_iterator</code>	Для итерации по каталогу
<code>recursive_directory_iterator</code>	Для итерации по каталогу и его подкаталогам

Рассмотрим простой, но не совсем уж нереальный пример:

```
void print_directory(path p)           // print the names of all files in p
try
{
    if (is_directory(p)) {
        cout << p << ":\n";
        for (const directory_entry& x : directory_iterator{p})
            cout << "    " << x.path() << '\n';
    }
}
catch (const filesystem_error& ex) {
    cerr << ex.what() << '\n';
}
```

Строка может быть неявно преобразована в `path`, поэтому мы можем использовать `print_directory` следующим образом:

```
void use()
{
    print_directory(".");           // current directory
    print_directory("..");          // parent directory
    print_directory("/");           // Unix root directory
    print_directory("c:");          // Windows volume C

    for (string s; cin>>s; )
        print_directory(s);
}
```

Если бы я хотел также перечислить подкаталоги, я бы сказал `recursive_directory_iterator{p}`. Если бы я хотел напечатать записи в лексикографическом порядке, я бы скопировал `path` в `vector` и отсортировал их перед печатью.

Класс `path` предлагает множество распространенных и полезных операций:

Операции с путями (некоторые)	
<code>p</code> и <code>p2</code> это <code>path</code>	
<code>value_type</code>	Тип символа, используемый собственной кодировкой файловой системы: <code>char</code> в POSIX, <code>wchar_t</code> в Windows
<code>string_type</code>	<code>std::basic_string<value_type></code>
<code>const_iterator</code>	<code>const</code> двунаправленный итератор с <code>value_type</code> для <code>path</code>
<code>iterator</code>	Псевдоним для <code>const_iterator</code>
<code>p=p2</code>	Присвоить <code>p2</code> в <code>p</code>
<code>p/=p2</code>	<code>p</code> и <code>p2</code> объединить с использованием разделителя (по умолчанию <code>/</code>)
<code>p+=p2</code>	<code>p</code> и <code>p2</code> объединить (без разделителя)
<code>s=p.native()</code>	Ссылка на нативный формат <code>p</code>

<code>s=p.string()</code>	<code>p</code> в нативном формате в виде <code>string</code>
<code>s=p.generic_string()</code>	<code>p</code> в общем формате в виде <code>string</code>
<code>p2=p.filename()</code>	Часть пути <code>p</code> включающая только имя файла
<code>p2=p.stem()</code>	Часть пути <code>p</code> включающая только каталоги
<code>p2=p.extension()</code>	Часть пути <code>p</code> включающая расширение файла
<code>i=p.begin()</code>	Начальный итератор последовательности элементов <code>p</code>
<code>i= p.end()</code>	Конечный итератор последовательности элементов <code>p</code>
<code>p==p2, p!=p2</code>	Проверка на равенство и неравенство путей <code>p</code> и <code>p2</code>
<code>p<p2, p<=p2, p>p2, p>=p2</code>	Лексикографическое сравнение
<code>is>>p, os<<p</code>	Потоковый I/O в/из <code>p</code>
<code>u8path(s)</code>	Путь из источника <code>s</code> , в кодировке UTF-8

Например:

```
void test(path p)
{
    if (is_directory(p)) {
        cout << p << ":\n";
        for (const directory_entry& x : directory_iterator(p)) {
            const path& f = x;           // refer to the path part of a directory en-
try
            if (f.extension() == ".exe")
                cout << f.stem() << " is a Windows executable\n";
            else {
                string n = f.extension().string();
                if (n == ".cpp" || n == ".C" || n == ".cxx")
                    cout << f.stem() << " is a C++ source file\n";
            }
        }
    }
}
```

Мы используем `path` в качестве строки (например, `f.extension`), и мы можем извлекать строки различных типов из `path` (например, `f.extension().string()`).

Соглашения об именовании, естественные языки и строковые кодировки отличаются высокой сложностью. Абстракции файловой системы стандартной библиотеки обеспечивают переносимость и значительное упрощение.

11.9.2 Файлы и каталоги

Естественно, файловая система предлагает множество операций, и естественно, что разные операционные системы предлагают разные наборы операций. Стандартная библиотека предлагает несколько вариантов, которые могут быть разумно реализованы в самых разных системах.

Операции с файловой системой (некоторые)	
<code>p</code> , <code>p1</code> , и <code>p2</code> это <code>path</code> ; <code>e</code> это <code>error_code</code> ; <code>b</code> это <code>bool</code> показывающий успешное или ошибочное завершение операции	
<code>exists(p)</code>	Ссылается ли <code>p</code> на существующий объект файловой системы?
<code>copy(p1,p2)</code>	Копирование файлов или каталогов из <code>p1</code> в <code>p2</code> ; сообщает об ошибках исключениями

<code>copy(p1,p2,e)</code>	Копирование файлов или каталогов; сообщает об ошибках при помощи кодов ошибок
<code>b=copy_file(p1,p2)</code>	Копирование файлов из <code>p1</code> в <code>p2</code> ; сообщает об ошибках исключениями
<code>b=create_directory(p)</code>	Создать новый каталог с именем <code>p</code> ; все промежуточные каталоги до <code>p</code> должны существовать
<code>b=create_directories(p)</code>	Создать новый каталог с именем <code>p</code> ; создаёт все промежуточные каталоги до <code>p</code>
<code>p=current_path()</code>	<code>p</code> присваивается текущий рабочий каталог
<code>current_path(p)</code>	Сделать <code>p</code> текущим рабочим каталогом
<code>s=file_size(p)</code>	<code>s</code> присваивается количество байт в <code>p</code>
<code>b=remove(p)</code>	Удалить <code>p</code> если это файл или пустой каталог

Многие операции имеют перегрузки, которые требуют дополнительных аргументов, таких как разрешения операционной системы. Работа с ними выходит далеко за рамки данной книги, поэтому ознакомьтесь с ними, если они вам понадобятся.

Как и `copy()`, все операции выполняются в двух вариантах:

- Базовая версия, указанная в таблице, например, `exists(p)`. Функция бросит исключение `filesystem_error`, если операция завершилась неудачей.
- Версия с дополнительным аргументом `error_code`, например, `exists(p,e)`. Проверьте `e`, чтобы убедиться, что операции выполнены успешно.

Мы используем коды ошибок, когда ожидается частый сбой операций при нормальном использовании, и операции с исключениями, когда ошибка считается редкой.

Часто использование функции запроса является самым простым и понятным подходом к изучению свойств файла. Библиотека `<filesystem>` знает о нескольких распространенных типах файлов и классифицирует остальные как “другие”:

Типы файлов	
<code>f</code> это <code>path</code> или <code>file_status</code>	
<code>is_block_file(f)</code>	Является ли <code>f</code> блочным устройством?
<code>is_character_file(f)</code>	Является ли <code>f</code> символьным устройством?
<code>is_directory(f)</code>	Является ли <code>f</code> каталогом?
<code>is_empty(f)</code>	Является ли <code>f</code> пустым файлом или каталогом?
<code>is_fifo(f)</code>	Является ли <code>f</code> именованным каналом?
<code>is_other(f)</code>	Является ли <code>f</code> каким-то другим видом файла?
<code>is_regular_file(f)</code>	Является ли <code>f</code> обычным (ординарным) файлом?
<code>is_socket(f)</code>	Является ли <code>f</code> именованным ИРС-сокетом?
<code>is_symlink(f)</code>	Является ли <code>f</code> символической ссылкой?
<code>status_known(f)</code>	Известен ли статус файла <code>f</code> ?

11.10 Советы

- [1] `iostream` являются типобезопасными, чувствительными к типу и расширяемыми; §11.1.
- [2] Используйте ввод на уровне символов только в тех случаях, когда это необходимо; §11.3; [CG: SL.io.1].
- [3] При чтении всегда учитывайте неправильно сформированный ввод; §11.3; [CG: SL.io.2].

- [4] Избегайте `endl` (если вы не знаете, что такое `endl`, вы ничего не пропустили); [CG: SL.io.50].
- [5] Определите `<<` и `>>` для пользовательских типов со значениями, которые имеют значимые текстовые представления; §11.1, §11.2, §11.3.
- [6] Используйте `cout` для нормального вывода и `cerr` для ошибок; §11.1.
- [7] Существуют потоки `iostream` для обычных символов и широких символов, и вы можете определить `iostream` для любого типа символов; §11.1.
- [8] Поддерживается двоичный ввод-вывод; §11.1.
- [9] Существуют стандартные `iostream` для стандартных потоков ввода-вывода, файлов и `string`; §11.2, §11.3, §11.7.2, §11.7.3.
- [10] Используйте цепочку операций `<<` для более краткой записи; §11.2.
- [11] Используйте цепочку операций `>>` для более краткой записи; §11.3.
- [12] Ввод в `string` не приводит к переполнению; §11.3.
- [13] Оператор `>>` по умолчанию пропускает начальный пробел; §11.3.
- [14] Используйте состояния потока `fail` для обработки потенциально исправимых ошибок ввода-вывода; §11.4.
- [15] Мы можем определить операторы `<<` и `>>` для пользовательских типов; §11.5.
- [16] Нам не нужно изменять `istream` или `ostream` для добавления новых операторов `<<` и `>>`; §11.5.
- [17] Используйте манипуляторы или `format()` для управления форматированием; §11.6.1, §11.6.2.
- [18] спецификации `precision()` применяются ко всем следующим операциям вывода с плавающей запятой; §11.6.1.
- [19] Спецификации формата для чисел с плавающей запятой (например, `scientific`) применяются ко всем следующим операциям вывода с плавающей; §11.6.1.
- [20] `#include <ios>` или `<iostream>` при использовании стандартных манипуляторов; §11.6.
- [21] Манипуляторы форматирования потока являются “липкими” для использования со последующими значениями в потоке; §11.6.1.
- [22] `#include <iomanip>` при использовании стандартных манипуляторов, принимающих аргументы; §11.6.
- [23] Мы можем выводить время, даты и т.д. в стандартных форматах; §11.6.1, §11.6.2.
- [24] Не пытайтесь скопировать поток: потоки только перемещаются; §11.7.
- [25] Не забудьте проверить, прикреплен ли файловый поток к файлу, прежде чем использовать его; §11.7.2.
- [26] Используйте `stringstream` или потоки памяти для форматирования в памяти; §11.7.3; §11.7.4.
- [27] Мы можем определить преобразования между любыми двумя типами, которые оба имеют строковое представление; §11.7.3.
- [28] Ввод-вывод в стиле C не является типобезопасным; §11.8.
- [29] Если вы не используете функции семейства `printf`, вызывайте `ios_base::sync_with_stdio(false)`; §11.8; [CG: SL.io.10].
- [30] Предпочтительнее использование `<filesystem>` прямому использованию интерфейсов, зависящих от платформы; §11.9.

Контейнеры

Это было ново. Это было необычно. Это было просто.

Это должно быть успешным!
– H. Nelson

- [Введение](#)
- `vector`

[Элементы](#); [Проверка диапазона](#)

- `list`
- `forward_list`
- `map`
- `unordered_map`
- [Аллокаторы](#)
- [Обзор контейнеров](#)
- [Советы](#)

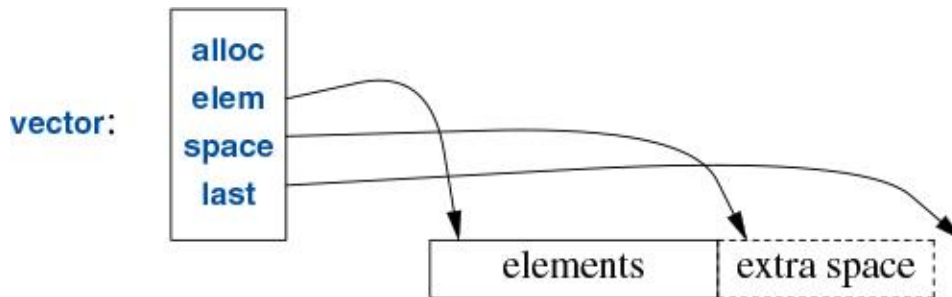
12.1 Введение

Большая часть вычислений включает в себя создание коллекций значений и последующее манипулирование такими коллекциями. Простой пример это чтение символов в `string` и вывод `string`. Класс, основной целью которого является хранение объектов, обычно называется *контейнером*. Предоставление подходящих контейнеров для данной задачи и поддержка контейнеров полезными базовыми операциями являются важными шагами при построении любой программы.

Чтобы проиллюстрировать контейнеры стандартной библиотеки, рассмотрим простую программу для хранения имен и телефонных номеров. Это та разновидность программ, для которой различные подходы кажутся “простыми и очевидными” для людей с разным опытом. Класс `Entry` из §11.5 можно использовать для хранения простой записи в телефонной книге. Здесь мы намеренно игнорируем многие сложности реального мира, такие как тот факт, что многие телефонные номера не имеют простого представления в виде 32-битного `int`.

12.2 `vector`

Наиболее полезным контейнером стандартной библиотеки является `vector`. `vector` - это последовательность элементов заданного типа. Элементы хранятся в памяти последовательно. Типичная реализация `vector` (§5.2.2, §6.2) будет состоять из дескриптора, содержащего указатели на первый элемент, на элемент следующий за последним и на элемент, следующий за выделенной памятью (§13.1) (или эквивалентную информацию, представленную в виде указателя плюс смещения):



Кроме того, он содержит аллокатор (здесь `alloc`), от которого `vector` может получать память для своих элементов. По умолчанию аллокатор использует `new` и `delete` для получения и освобождения памяти (§12.7). Используя немного продвинутый метод реализации, мы можем избежать хранения каких-либо данных для простых аллокаторов в объекте `vector`.

Мы можем инициализировать `vector` набором значений, соответствующих типу его элемента:

```
vector<Entry> phone_book = {
    {"David Hume",123456},
    {"Karl Popper",234567},
    {"Bertrand Arthur William Russell",345678}
};
```

Доступ к элементам возможен через индекс. Итак, предполагая, что мы определили << для `Entry`, мы можем написать:

```
void print_book(const vector<Entry>& book)
{
    for (int i = 0; i!=book.size(); ++i)
        cout << book[i] << '\n';
}
```

Как обычно, индексация начинается с 0, так что в `book[0]` содержится запись для `David Hume`. Функция-член `size()` возвращает количество элементов.

Элементы `vector` образуют диапазон, поэтому мы можем использовать цикл `for` для диапазонов (§1.7):

```
void print_book(const vector<Entry>& book)
{
    for (const auto& x : book)           // for "auto" see §1.4
        cout << x << '\n';
}
```

Когда мы определяем `vector`, мы задаем ему начальный размер (начальное количество элементов):

```
vector<int> v1 = {1, 2, 3, 4};           // size is 4
vector<string> v2;                       // size is 0
vector<Shape*> v3(23);                   // size is 23; initial element value: nullptr
vector<double> v4(32,9.9);              // size is 32; initial element value: 9.9
```

Явно указанный размер заключен в обычные круглые скобки, например, (23), и обычно элементы инициализируются значением по умолчанию соответственно типу элемента (например, `nullptr` для указателей и `0` для чисел). Если вам не нужно значение по умолчанию, вы можете указать начальное значение всех элементов в качестве второго аргумента (например, `9.9` для `32` элементов `v4`).

Начальный размер может быть изменен. Одной из наиболее полезных операций с `vector` является `push_back()`, которая добавляет новый элемент в конец вектора, увеличивая его размер на единицу. Например, предполагая, что мы определили `>>` для `Entry`, мы можем написать:

```
void input()
{
    for (Entry e; cin>>e; )
        phone_book.push_back(e);
}
```

Здесь считываем `Entry` из стандартного ввода в `phone_book` до тех пор, пока либо не будет достигнут конец ввода (например, конец файла), либо операция ввода не столкнется с ошибкой формата.

`vector` стандартной библиотеки реализован таким образом, что увеличение `vector` с помощью повторных `push_back()` является эффективным. Чтобы показать, как это делается, рассмотрим разработку простого `Vector` из [главы 5](#) и [главы 7](#), используя представление, указанное на диаграмме выше:

```
template<typename T>
class Vector {
    allocator<T> alloc;           // standard-library allocator of space for Ts
    T* elem;                      // pointer to first element
    T* space;                     // pointer to first unused (and uninitialized) slot
    T* last;                      // pointer to last slot
public:
    // ...
    int size() const { return space - elem; } // number of elements
    int capacity() const { return last - elem; } // number of slots available for elements
    // ...
    void reserve(int newsz); // increase capacity() to newsz
    // ...
    void push_back(const T& t); // copy t into Vector
    void push_back(T&& t); // move t into Vector
};
```

`vector` стандартной библиотеки имеет методы `capacity()`, `reserve()` и `push_back()`. Функция `reserve()` используется пользователями `vector` и другими методами `vector`, чтобы зарезервировать место в памяти для большего количества элементов. Возможно, ему придется выделить новую память, и когда это произойдет, он переместит элементы в новое место. Когда функция `reserve()` перемещает элементы в новое, более крупное место, старые указатели на эти элементы теперь будут указывать на неправильное местоположение; они становятся *недействительными* и не должны использоваться.

Учитывая `capacity()` и `reserve()`, реализация `push_back()` является тривиальной:

```
template<typename T>
void Vector<T>::push_back(const T& t)
{
    if (capacity() <= size()) // make sure we have space for t
        reserve(size() * 2); // double the capacity
    construct_at(space, t); // initialize *space to t ("place t at space")
}
```

```

    ++space;
}

```

Теперь выделение и перемещение элементов происходит нечасто. Раньше я использовал `reserve()`, чтобы попытаться повысить производительность, но это оказалось пустой тратой усилий: эвристика, используемая `vector`, в среднем лучше, чем мои предположения, поэтому теперь я явно использую `reserve()` только для того, чтобы избежать переаллокации элементов, когда я хочу использовать указатели на элементы.

`vector` может быть скопирован при присвоении и инициализации. Например:

```
vector<Entry> book2 = phone_book;
```

Копирование и перемещение `vector` реализуются конструкторами и операторами присваивания, как описано в §6.2. Присваивание вектора предполагает копирование его элементов. Таким образом, после инициализации `book2`, `book2` и `phone_book` хранят отдельные копии каждой `Entry` в телефонной книге. Когда `vector` содержит много элементов, такие выглядящие невинно присваивания и инициализации могут быть дорогостоящими. Там, где копирование нежелательно, следует использовать ссылки или указатели (§1.7) или операции перемещения (§6.2.2).

`vector` стандартной библиотеки очень гибкий и эффективный. Используйте его в качестве контейнера по умолчанию; то есть используйте его, если у вас нет веской причины использовать какой-либо другой контейнер. Если вы избегаете `vector` из-за смутных опасений по поводу “эффективности”, измерьте его быстродействие сами. Наша интуиция чаще всего ошибается в вопросах эффективности использования контейнеров.

12.2.1 Элементы

Как и все контейнеры стандартной библиотеки, `vector` - это контейнер элементов некоторого типа `T`, то есть `vector<T>`. Практически любой тип квалифицируется как тип элемента: встроенные числовые типы (такие как `char`, `int` и `double`), пользовательские типы (такие как `string`, `Entry`, `list<int>` и `Matrix<double,2>`) и указатели (такие как `const char*`, `Shape*` и `double*`). Когда вы вставляете новый элемент, его значение копируется в контейнер. Например, когда вы помещаете целое число со значением `7` в контейнер, результирующий элемент действительно имеет значение `7`. Элемент не является ссылкой или указателем на какой-либо объект, содержащий `7`. Это позволяет создавать красивые, компактные контейнеры с быстрым доступом. Для людей, которые заботятся об объемах памяти и производительности во время выполнения, это очень важно.

Если у вас есть иерархия классов (§5.5), которая полагается на виртуальные функции для получения полиморфного поведения, не храните объекты непосредственно в контейнере. Вместо этого храните указатель (или умный указатель; §15.2.1). Например:

```
vector<Shape> vs;           // No, don't - there is no room for a Circle or a Smiley (§5.5)
vector<Shape*> vps;         // better, but see §5.5.3 (don't Leak)
vector<unique_ptr<Shape>> vups; // OK

```

12.2.2 Проверка диапазона

`vector` стандартной библиотеки не гарантирует проверку диапазона. Например:

```
void silly(vector<Entry>& book)
{
    int i = book[book.size()].number;           // book.size() is out of range

```

```

// ...
}

```

Эта инициализация, скорее всего, поместит некоторое случайное значение в `i`, а не выдаст ошибку. Это нежелательно, и ошибки, связанные с выходом за пределы диапазона, являются распространенной проблемой. Поэтому, я часто использую простую адаптацию `vector` для проверки диапазона:

```

template<typename T>
struct Vec : std::vector<T> {
    using vector<T>::vector;    // use the constructors from vector (under the name
Vec)

    T& operator[](int i) { return vector<T>::at(i); }    // range check
    const T& operator[](int i) const { return vector<T>::at(i); }    // range check
const
                                                                    // objects;

```

§5.2.1

```

    auto begin() { return Checked_iter<vector<T>>{*this}; }    // see §13.1
    auto end() { return Checked_iter<vector<T>>{*this, vector<T>::end()}; }
};

```

`Vec` наследует все от `vector`, за исключением операций с индексами, которые он переопределяет для проверки диапазона. Операция `at()` - это операция индекса `vector`, которая генерирует исключение типа `out_of_range`, если ее аргумент находится вне диапазона `vector` (§4.2).

Для `Vec` доступ вне диапазона вызовет исключение, которое пользователь может перехватить. Например:

```

void checked(Vec<Entry>& book)
{
    try {
        book[book.size()] = {"Joe", 999999};    // will throw an exception
        // ...
    }
    catch (out_of_range&) {
        cerr << "range error\n";
    }
}

```

Исключение будет брошено, а затем перехвачено (§4.2). Если пользователь не перехватит исключение, программа завершится четко определенным образом, а не продолжится или завершится с ошибкой неопределенным образом. Один из способов свести к минимуму неожиданности от неперехваченных исключений - использовать `main()` с `try-block` в качестве тела. Например:

```

int main()
try {
    // your code
}

catch (out_of_range&) {
    cerr << "range error\n";
}
catch (...) {
    cerr << "unknown exception thrown\n";
}

```

Этот код предоставляет обработчики исключений по умолчанию, так что, если нам не удастся перехватить какое-либо исключение, выводится сообщение об ошибке в стандартном потоке вывода ошибок `cerr` (§11.2).

Почему стандарт не гарантирует проверку диапазона? Многие приложения, для которых критически важна производительность, используют `vector`, и проверка всех индексов влечет за собой увеличение затрат примерно на 10%. Очевидно, что эта стоимость может сильно варьироваться в зависимости от аппаратного обеспечения, оптимизаторов и использования индексов приложением. Однако опыт показывает, что такие накладные расходы могут привести к тому, что люди предпочтут гораздо более небезопасные встроенные массивы. Даже простой страх перед такими накладными расходами может привести к отказу от использования `vector`. По крайней мере, диапазон `vector` легко проверяется во время отладки, и мы можем создавать проверенные версии поверх непроверенных по умолчанию.

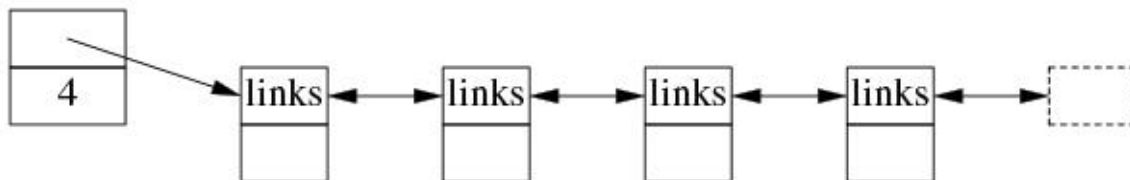
Цикл `for` для диапазонов позволяет избежать ошибок диапазона без каких-либо затрат за счет неявного доступа ко всем элементам в диапазоне. Пока их аргументы действительны, алгоритмы стандартной библиотеки гарантируют отсутствие ошибок диапазона.

Если вы используете `vector::at()` непосредственно в своем коде, вам не нужен мой обходной путь через `Vec`. Кроме того, некоторые стандартные библиотеки имеют реализации `vector` с проверкой диапазона, которые предлагают более полную проверку, чем `Vec`.

12.3 list

Стандартная библиотека предлагает двусвязный список, называемый `list`:

`list`:



Мы используем `list` для последовательностей, в которые мы хотим вставлять и удалять элементы, не перемещая другие элементы. Вставка и удаление записей телефонной книги может быть обычным делом, поэтому `list` может быть подходящим для представления простой телефонной книги. Например:

```
list<Entry> phone_book = {
    {"David Hume", 123456},
    {"Karl Popper", 234567},
    {"Bertrand Arthur William Russell", 345678}
};
```

Когда мы используем двусвязанный список, мы, как правило, не получаем доступ к элементам, используя индекс, как мы обычно делаем для векторов. Вместо этого мы могли бы выполнить поиск по списку в поисках элемента с заданным значением. Чтобы сделать это, мы воспользуемся тем фактом, что `list` - это последовательность, как описано в [главе 13](#):

```
int get_number(const string& s)
{
    for (const auto& x : phone_book)
        if (x.name==s)
            return x.number;
```

```

    return 0; // use 0 to represent "number not found"
}

```

Поиск `s` начинается с начала списка и продолжается до тех пор, пока не будет найден `s` или не будет достигнут конец `phone_book`.

Иногда нам нужно идентифицировать элемент в `list`. Например, мы можем захотеть удалить элемент или вставить перед ним новый элемент. Для этого мы используем *итератор*: итератор `list` указывает на элемент `list` и может использоваться для итерации по `list` (отсюда и его название). Каждый контейнер стандартной библиотеки предоставляет функции `begin()` и `end()`, которые возвращают итератор на первый и на следующий за последним элемент соответственно (§13.1). Используя итераторы явно, мы можем – менее элегантно – написать функцию `get_number()` следующим образом

```

int get_number(const string& s)
{
    for (auto p = phone_book.begin(); p!=phone_book.end(); ++p)
        if (p->name==s)
            return p->number;
    return 0; // use 0 to represent "number not found"
}

```

На самом деле, примерно так компилятор реализует более краткий и менее подверженный ошибкам цикл `for` для диапазона. Учитывая что `p` это итератор, `*p` – это элемент, на который он ссылается, `++p` передвигает `p` для ссылки на следующий элемент, и когда `p` ссылается на класс с членом `m`, тогда `p->m` эквивалентно `(*p).m`.

Добавлять элементы в `list` и удалять элементы из `list` очень просто:

```

void f(const Entry& ee, list<Entry>::iterator p, list<Entry>::iterator q)
{
    phone_book.insert(p,ee); // add ee before the element referred to by p
    phone_book.erase(q);    // remove the element referred to by q
}

```

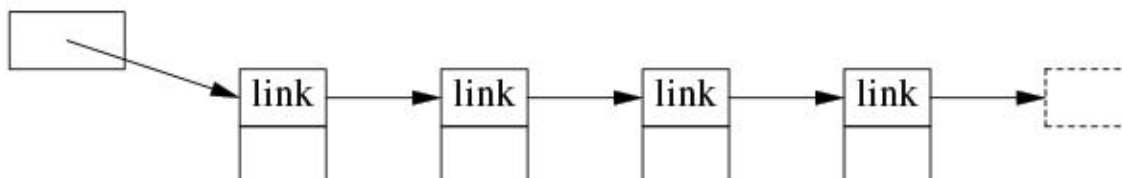
Для `list` вызов `insert(p, elem)` вставляет элемент с копией значения `elem` перед элементом, на который указывает `p`. Здесь `p` может быть итератором, указывающим на следующий за последним элементом `list`. И наоборот, `erase(p)` удаляет элемент, на который указывает `p`, и уничтожает его.

Эти примеры `list` могут быть написаны идентично с использованием `vector` и (удивительно, если вы не разбираетесь в архитектуре машины) часто работают лучше с `vector`, чем со `list`. Когда все, что нам нужно, – это последовательность элементов, у нас есть выбор между использованием `vector` и `list`. Если у вас нет причин поступить иначе, используйте `vector`. `vector` лучше подходит для обхода (например, `find()` и `count()`), а также для сортировки и поиска (например, `sort()` и `equal_range()`; §13.5, §15.3.3).

12.4 forward_list

Стандартная библиотека также предлагает односвязный список под названием `forward_list`:

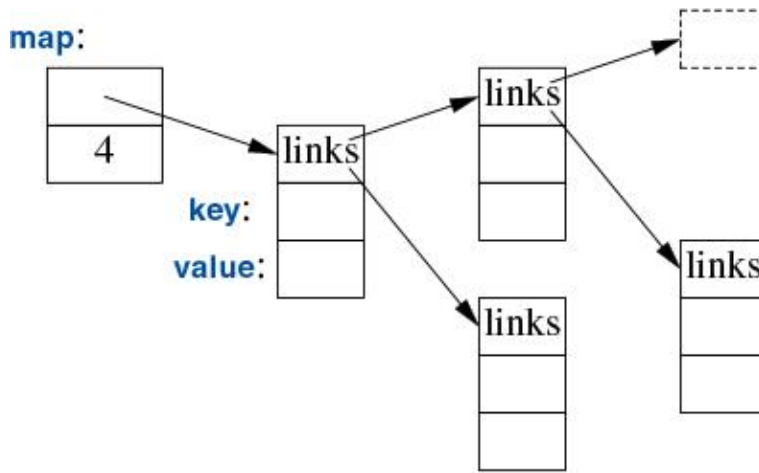
`forward_list`:



Список `forward_list` отличается от (двусвязного) `list` тем, что разрешает только прямую итерацию. Смысл этого в том, чтобы сэкономить место. Нет необходимости сохранять указатель-на элемент предшественник в каждой ссылке, а размер пустого списка `forward_list` равен всего одному указателю. `forward_list` даже не сохраняет количество своих элементов. Если вам нужно количество элементов, подсчитайте. Если вы не можете позволить себе считать, вам, вероятно, не следует использовать `forward_list`.

12.5 map

Написание кода для поиска имени в списке пар (*name, number*) довольно утомительно. Кроме того, линейный поиск неэффективен для всех списков, кроме самых коротких. Стандартная библиотека предлагает сбалансированное двоичное дерево поиска (обычно красно-черное дерево), называемое `map`:



В других контекстах `map` известна как ассоциативный массив или словарь.

`map` стандартной библиотеки - это контейнер пар значений, оптимизированный для поиска и вставки. Мы можем использовать тот же список инициализации, что и для `vector` и `list` (§12.2, §12.3):

```
map<string,int> phone_book {
    {"David Hume",123456},
    {"Karl Popper",234567},
    {"Bertrand Arthur William Russell",345678}
};
```

При использовании в качестве индекса значения своего первого типа (называемого *ключом*) `map` возвращает соответствующее значение второго типа (называемое *значением* или *сопоставленным типом*). Например:

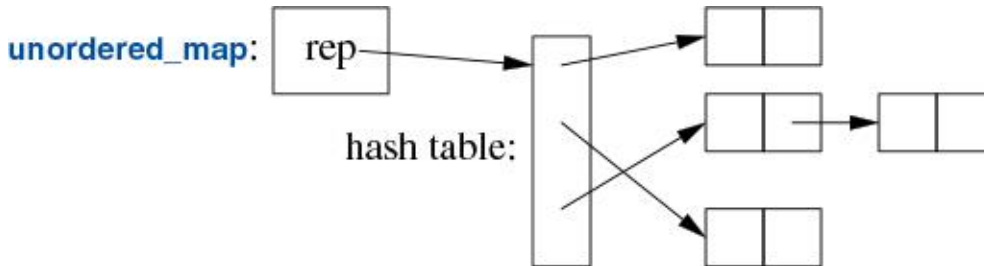
```
int get_number(const string& s)
{
    return phone_book[s];
}
```

Другими словами, индексация по `map` - это, по сути, поиск, который мы называли `get_number()`. Если `key` не найден, он вводится в `map` со значением по умолчанию для его `value`. Значение по умолчанию для целочисленного типа равно 0, и это просто разумное значение для представления недопустимого телефонного номера.

Если бы мы хотели избежать ввода недопустимых номеров в нашу телефонную книгу, мы могли бы использовать `find()` и `insert()` (§12.8) вместо `[]`.

12.6 unordered_map

Стоимость поиска по `map` равна $O(\log(n))$, где n - количество элементов в `map`. Это довольно хорошо. Например, для `map` с 1 000 000 элементов мы выполняем всего около 20 сравнений и косвенных указаний, чтобы найти элемент. Однако во многих случаях мы можем добиться большего успеха, используя поиск по хэшу, а не сравнение с использованием функции упорядочения, такой как `<`. Хэшированные контейнеры стандартной библиотеки называются “неупорядоченными”, поскольку для них не требуется функция упорядочивания:



Например, мы можем использовать `unordered_map` из `<unordered_map>` для нашей телефонной книги:

```
unordered_map<string,int> phone_book {
    {"David Hume",123456},
    {"Karl Popper",234567},
    {"Bertrand Arthur William Russell",345678}
};
```

Как и в случае с `map`, мы можем использовать оператор индекса для `unordered_map`:

```
int get_number(const string& s)
{
    return phone_book[s];
}
```

Стандартная библиотека предоставляет для `string` хэш-функцию по умолчанию, а также для других встроенных типов и типов стандартной библиотеки. При необходимости мы можем предоставить свои собственные. Возможно, наиболее распространенная потребность в пользовательской хэш-функции возникает, когда нам нужен неупорядоченный контейнер одного из наших собственных типов. Хэш-функция часто реализуется как функциональный объект (§7.3.2). Например:

```
struct Record {
    string name;
    int product_code;
    // ...
};

struct Rhash {                // a hash function for Record
    size_t operator()(const Record& r) const
    {
        return hash<string>()(r.name) ^ hash<int>()(r.product_code);
    }
};

unordered_set<Record,Rhash> my_set; // set of Records using Rhash for lookup
```

Разработка хороших хэш-функций - это искусство, и часто требуется знания данных, к которым они будут применяться. Создание новой хэш-функции путем объединения

существующих хэш-функций с использованием исключающего ИЛИ (^) является простым и часто очень эффективным. Однако будьте осторожны, чтобы убедиться, что каждое значение, участвующее в хэше, действительно помогает различать значения. Например, если у вас не может быть нескольких названий для одного и того же кода продукта (или нескольких кодов продукта для одного и того же имени), объединение двух хэшей не дает никаких преимуществ.

Мы можем избежать явной передачи операции `hash`, определив ее как специализацию `hash` стандартной библиотеки:

```
namespace std {                                     // make a hash function for Record

    template<> struct hash<Record> {
        using argument_type = Record;
        using result_type = size_t;

        result_type operator()(const Record& r) const
        {
            return hash<string>()(r.name) ^ hash<int>()(r.product_code);
        }
    };
}
```

Обратите внимание на различия между `map` и `unordered_map`:

- Для `map` требуется функция упорядочивания (по умолчанию используется `<`), которая выдает упорядоченную последовательность.
- Для `unordered_map` требуется функция равенства (по умолчанию `==`); она не поддерживает порядок между своими элементами.

Учитывая хорошую хэш-функцию, `unordered_map` работает намного быстрее, чем `map` для больших контейнеров. Однако в худшем случае поведение `unordered_map` с плохой хэш-функцией намного хуже, чем у `map`.

12.7 Аллокаторы

По умолчанию контейнеры стандартной библиотеки выделяют пространство с помощью `new`. Операторы `new` и `delete` предоставляют динамическую память (также называемую кучей), в которой могут храниться объекты произвольного размера и контролируемого пользователем срока жизни. Это влечет за собой накладные расходы времени и памяти, которые могут быть устранены во многих особых случаях. Таким образом, контейнеры стандартной библиотеки предоставляют возможность устанавливать аллокаторы с определенной семантикой там, где это необходимо. Специальные аллокаторы используются для решения широкого спектра проблем, связанных с производительностью (например, пул аллокаторов), безопасностью (аллокаторы, которые очищают память при удалении), аллокацией по потокам и неоднородными архитектурами памяти (аллокация в определенных областях памяти с соответствующими типами указателей). Здесь не место обсуждать эти важные, но очень специализированные и часто продвинутые методы. Однако я приведу один пример, подкреплённый реальной проблемой, решением которой был пул аллокаторов.

Важная, долго работающая система использовала очередь событий (см. §18.4), используя `vector` в качестве объекта событий, которые передавались как `shared_ptr`. Таким образом, последний пользователь события неявно удалил бы его:

```
struct Event {
    vector<int> data = vector<int>(512);
};
```

```
list<shared_ptr<Event>> q;

void producer()
{
    for (int n = 0; n!=LOTS; ++n) {
        lock_guard lk {m};           // m is a mutex; see §18.3
        q.push_back(make_shared<Event>());
        cv.notify_one();             // cv is a condition_variable; see §18.4
    }
}
```

С логической точки зрения это сработало прекрасно. Это логически просто, поэтому код надежен и удобен в обслуживании. К сожалению, это привело к массовой фрагментации. После того, как 100 000 событий были переданы 16 производителям и 4 потребителям, было израсходовано более 6 ГБ памяти.

Традиционным решением проблем фрагментации является переписывание кода для использования пула аллокаторов. Пул аллокаторов - это аллокатор, который управляет объектами одного фиксированного размера и выделяет пространство для многих объектов одновременно, а не использует отдельные аллокации. К счастью, C++ предлагает прямую поддержку для этого. Пул аллокаторов определен в подпространстве имен **pmr** (“полиморфный ресурс памяти”) пространства имён **std**:

```
pmr::synchronized_pool_resource pool;           // make a pool

struct Event {
    vector<int> data = vector<int>{512,&pool};    // Let Events use the pool
};

list<shared_ptr<Event>> q {&pool};               // Let q use the pool

void producer()
{
    for (int n = 0; n!=LOTS; ++n) {
        scoped_lock lk {m};                     // m is a mutex (§18.3)
        q.push_back(allocate_shared<Event,pmr::polymorphic_allocator<Event>>(&pool));
        cv.notify_one();
    }
}
```

Теперь, после того, как 100 000 событий были переданы 16 производителям и 4 потребителям, было израсходовано менее 3 МБ памяти. Это оптимизация примерно в 2000 раз! Естественно, объем фактически используемой памяти (в отличие от памяти, потраченной впустую из-за фрагментации) остается неизменным. После устранения фрагментации использование памяти со временем стало стабильным, так что система могла работать в течение нескольких месяцев.

Методы, подобные этому, применялись с хорошими результатами с первых дней существования C++, но, как правило, они требовали переписывания кода для использования специализированных контейнеров. Теперь стандартные контейнеры опционально принимают аргументы аллокатора. По умолчанию контейнеры используют **new** и **delete**. Другие ресурсы полиморфной памяти включают

- **unsynchronized_polymorphic_resource**; похож на **polymorphic_resource** но может использоваться только одним потоком.
- **monotonic_polymorphic_resource**; быстрый аллокатор, который освобождает свою память только при ее уничтожении и может использоваться только одним потоком.

Полиморфный ресурс должен быть получен из `memory_resource` и определять элементы `allocate()`, `deallocate()` и `is_equal()`. Идея заключается в том, чтобы пользователи создавали свои собственные ресурсы для настройки кода.

12.8 Обзор контейнеров

Стандартная библиотека предоставляет некоторые из наиболее общих и полезных типов контейнеров, позволяющих программисту выбрать контейнер, который наилучшим образом соответствует потребностям приложения:

Краткое описание стандартных контейнеров	
<code>vector<T></code>	Динамический массив (§12.2)
<code>list<T></code>	Двусвязный список (§12.3)
<code>forward_list<T></code>	Односвязный список
<code>deque<T></code>	Двусторонняя очередь
<code>map<K,V></code>	Ассоциативный массив (§12.5)
<code>multimap<K,V></code>	Карта, в которой ключ может встречаться много раз
<code>unordered_map<K,V></code>	Карта, использующая поиск на основе хэша (§12.6)
<code>unordered_multimap<K,V></code>	Multimap с использованием поиска на основе хэша
<code>set<T></code>	Набор (карта только с ключом и без значения)
<code>multiset<T></code>	Набор, в котором ключ может встречаться много раз
<code>unordered_set<T></code>	Набор, использующий поиск на основе хэша
<code>unordered_multiset<T></code>	Multiset, использующий поиск на основе хэша

Неупорядоченные контейнеры оптимизированы для поиска по ключу (часто в виде строки); другими словами, они являются хэш-таблицами.

Контейнеры определены в пространстве имен `std` и представлены в заголовках `<vector>`, `<list>`, `<map>` и т.д. (§9.3.4). Кроме того, стандартная библиотека предоставляет адаптеры контейнеров `queue<T>`, `stack<T>` и `priority_queue<T>`. Изучите их, если они вам понадобятся. Стандартная библиотека также предоставляет более специализированные типы, подобные контейнерам, такие как `array<T,N>` (§15.3.1) и `bitset<N>` (§15.3.2).

Стандартные контейнеры и их основные операции спроектированы таким образом, чтобы быть похожими с точки зрения обозначения. Кроме того, смысл операций эквивалентен для различных контейнеров. Основные операции применимы ко всем типам контейнеров, для которых они имеют смысл и могут быть эффективно реализованы:

Операции стандартных контейнеров (некоторые)	
<code>value_type</code>	Тип элемента
<code>p=c.begin()</code>	<code>p</code> указывает на первый элемент в <code>c</code> ; также <code>cbegin()</code> для <code>const</code> итератора
<code>p=c.end()</code>	<code>p</code> указывает на элемент следующий за последним в <code>c</code> ; также <code>cend()</code> для <code>const</code> итератора
<code>k=c.size()</code>	<code>k</code> равно количеству элементов в <code>c</code>
<code>c.empty()</code>	Пуст ли <code>c</code> ?
<code>k=c.capacity()</code>	<code>k</code> это количество элементов которое <code>c</code> может хранить без новой аллокации
<code>c.reserve(k)</code>	Увеличивает ёмкость до <code>k</code> ; если <code>k<=c.capacity()</code> , <code>c.reserve(k)</code> ничего не делает
<code>c.resize(k)</code>	Устанавливает количество элементов <code>k</code> ; добавленные элементы имеют значение по умолчанию <code>value_type{} </code>

<code>c[k]</code>	<code>k</code> -й элемент в <code>c</code> ; индекс первого элемента 0; без гарантий проверки диапазона
<code>c.at(k)</code>	<code>k</code> -й элемент в <code>c</code> ; если выходит за диапазон, бросает исключение <code>out_of_range</code>
<code>c.push_back(x)</code>	Добавить <code>x</code> в конец <code>c</code> ; увеличивает размер <code>c</code> на единицу
<code>c.emplace_back(a)</code>	Добавить <code>value_type{a}</code> в конец <code>c</code> ; увеличивает размер <code>c</code> на единицу
<code>q=c.insert(p,x)</code>	Вставить <code>x</code> перед <code>p</code> в <code>c</code>
<code>q=c.erase(p)</code>	Удаляет элемент с индексом <code>p</code> из <code>c</code>
<code>c=c2</code>	Присвоение: копирует все элементы из <code>c2</code> для получения <code>c==c2</code>
<code>b=(c==c2)</code>	Равенство всех элементов <code>c</code> и <code>c2</code> ; <code>b==true</code> если элементы равны
<code>x=(c<=>c2)</code>	Лексикографическое сравнение <code>c</code> и <code>c2</code> : <code>x<0</code> если <code>c</code> меньше чем <code>c2</code> , <code>x==0</code> если они равны, и <code>0<x</code> если <code>c</code> больше чем <code>c2</code> . <code>!=, <, <=, > и >=</code> производные от <code><=></code>

Такое нотационное и семантическое единообразие позволяет программистам создавать новые типы контейнеров, которые могут использоваться очень похожим образом на стандартные. Например вектор с проверкой границ диапазона, `Vector` (§4.3, глава 5). Единообразие интерфейсов контейнеров позволяет нам определять алгоритмы независимо от отдельных типов контейнеров. Однако у каждого из них есть свои сильные и слабые стороны. Например, оператор индекса и перемещение по `vector` дешево и просто. С другой стороны, элементы `vector` переаллоцируются, когда мы вставляем или удаляем элементы; `list` обладает прямо противоположными свойствами. Пожалуйста, обратите внимание, что `vector` обычно более эффективен, чем `list`, для коротких последовательностей небольших элементов (даже для `insert()` и `erase()`). Я рекомендую `vector` стандартной библиотеки в качестве типа по умолчанию для последовательностей элементов: вам нужна веская причина, чтобы выбрать другой.

Рассмотрим односвязный список `forward_list`, контейнер, оптимизированный для пустой последовательности (§12.3). Пустой `forward_list` занимает всего одно слово (размер указателя в 32-разрядной системе), в то время как пустой `vector` занимает три. Пустые последовательности и последовательности, содержащие только один или два элемента, на удивление распространены и полезны.

Операция размещения, такая как `emplace_back()`, принимает аргументы для конструктора элемента и создает объект во вновь выделенном пространстве в контейнере, вместо копирования объекта в контейнер. Например, для `vector<pair<int,string>>` мы могли бы написать:

```
v.push_back(pair{1,"copy or move"});    // make a pair and move it into v
v.emplace_back(1,"build in place");    // build a pair in v
```

Для простых примеров, подобных этому, оптимизация может привести к эквивалентной производительности для обоих вызовов.

12.9 Советы

- [1] Контейнер STL определяет последовательность; §12.2.
- [2] Контейнеры STL - это дескрипторы ресурсов; §12.2, §12.3, §12.5, §12.6.
- [3] Используйте `vector` в качестве контейнера по умолчанию; §12.2, §12.8; [CG: SL.con.2].
- [4] Для простого обхода контейнера используйте цикл `for` для диапазона или пару итераторов `begin/end`; §12.2, §12.3.

- [5] Используйте `reserve()`, чтобы избежать аннулирования указателей и итераторов на элементы; §12.2.
- [6] Не предполагайте преимущества производительности от `reserve()` без измерения; §12.2.
- [7] Используйте `push_back()` или `resize()` в контейнере, вместо `realloc()` в массиве; §12.2.
- [8] Не используйте старые итераторы в `vector` после изменения размера; §12.2 [CG: ES.65].
- [9] Не думайте, что `[]` проверяет границы диапазона; §12.2.
- [10] Используйте `at()`, когда вам нужна гарантированная проверка границ диапазона; §12.2; [CG: SL.con.3].
- [11] Используйте `for` для диапазона и алгоритмы стандартной библиотеки для устранения ошибок диапазона без накладных расходов; §12.2.2.
- [12] Элементы копируются в контейнер; §12.2.1.
- [13] Чтобы сохранить полиморфное поведение элементов, храните указатели (встроенные или определяемые пользователем); §12.2.1.
- [14] Операции вставки, такие как `insert()` и `push_back()`, часто оказываются удивительно эффективными для `vector`; §12.3.
- [15] Используйте `forward_list` для последовательностей, которые обычно пусты; §12.8.
- [16] Когда дело доходит до производительности, не доверяйте своей интуиции: измеряйте; §12.2.
- [17] `map` обычно реализована в виде красно-черного дерева; §12.5.
- [18] `unordered_map` это хэш таблица; §12.6.
- [19] Передавайте контейнер по ссылке и возвращайте контейнер по значению; §12.2.
- [20] Для контейнера используйте синтаксис `()`-инициализатора для размеров и синтаксис `{}`-инициализатора для последовательностей элементов; §5.2.3, §12.2.
- [21] Предпочитайте компактные и непрерывные структуры данных; §12.3.
- [22] Обход `list` относительно дорогой; §12.3.
- [23] Используйте неупорядоченные контейнеры, если вам нужен быстрый поиск среди больших объемов данных; §12.6.
- [24] Используйте упорядоченные контейнеры (например, `map` и `set`), если вам нужно перебирать их элементы по порядку; §12.5.
- [25] Используйте неупорядоченные контейнеры (например, `unordered_map`) для типов элементов без естественного порядка (т.е. без разумного `<`); §12.5.
- [26] Используйте ассоциативные контейнеры (например, `map` и `list`), когда вам нужно, чтобы указатели на элементы были стабильными при изменении размера контейнера; §12.8.
- [27] Поэкспериментируйте, чтобы убедиться, что у вас есть приемлемая хэш-функция; §12.6.
- [28] Хэш-функция, полученная путем объединения стандартных хэш-функций для элементов с использованием оператора исключающее ИЛИ (`^`), часто хороша; §12.6.
- [29] Знайте контейнеры стандартной библиотеки и предпочитайте их структурам данных, созданным вручную; §12.8.
- [30] Если ваше приложение страдает от проблем с производительностью, связанных с памятью, сведите к минимуму использование динамической памяти

и/или рассмотрите возможность использования специализированного аллокатора; §[12.7](#).

Алгоритмы

*Не умножайте сущности сверх
необходимости.
– William Occam*

- [Введение](#)
- [Использование итераторов](#)
- [Типы итераторов](#)

[Потоковые итераторы](#)

- [Использование предикатов](#)
- [Обзор алгоритмов](#)
- [Параллельные алгоритмы](#)
- [Советы](#)

13.1 Введение

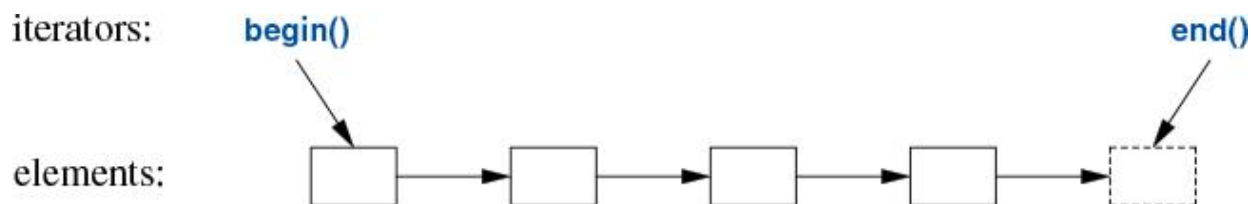
Структура данных, такая как список или вектор, сама по себе не очень полезна. Чтобы использовать их, нам нужны операции для базового доступа, такие как добавление и удаление элементов (как предусмотрено для `list` и `vector`). Кроме того, мы редко просто храним объекты в контейнере. Мы сортируем их, выводим, извлекаем подмножества, удаляем элементы, ищем объекты и т.д. Поэтому стандартная библиотека предоставляет наиболее распространенные алгоритмы для контейнеров в дополнение к наиболее распространенным типам контейнеров. Например, мы можем просто и эффективно отсортировать `vector` из `Entry` и поместить копию каждого уникального элемента `vector` в `list`:

```
void f(vector<Entry>& vec, list<Entry>& lst)
{
    sort(vec.begin(),vec.end());           // use < for order
    unique_copy(vec.begin(),vec.end(),lst.begin()); // don't copy adjacent equal ele-
ments
}
```

Чтобы этот код сработал, для `Entry` должны быть определены операторы меньше (`<`) и равно (`==`). Например:

```
bool operator<(const Entry& x, const Entry& y)    // Less than
{
    return x.name<y.name;                        // order Entries by their names
}
```

Стандартный алгоритм выражается в терминах (полуоткрытых) последовательностей элементов. *Последовательность* представлена парой итераторов, задающих первый элемент и элемент следующий за последним:



В примере функция `sort()` сортирует последовательность, определенную парой итераторов `vec.begin()` и `vec.end()`, которая ограничивает все элементы `vector`. Для записи (вывода) нам нужно только указать первый записываемый элемент. Если записано более одного элемента, элементы, следующие за этим начальным элементом, будут перезаписаны. Таким образом, чтобы избежать ошибок, `lst` должен содержать по крайней мере столько элементов, сколько уникальных значений в `vec`.

К сожалению, стандартная библиотека не предлагает абстракцию для поддержки записи в контейнер с проверкой диапазона. Однако мы можем определить один:

```
template<typename C>
class Checked_iter {
public:
    using value_type = typename C::value_type;
    using difference_type = int;

    Checked_iter() { throw Missing_container{}; } // concept forward_iterator requires
a                                     // default constructor

    Checked_iter(C& cc) : pc{ &cc } {}
    Checked_iter(C& cc, typename C::iterator pp) : pc{ &cc }, p{ pp } {}

    Checked_iter& operator++() { check_end(); ++p; return *this; }
    Checked_iter operator++(int) { check_end(); auto t{ *this }; ++p; return t; }
    value_type& operator*() const { check_end(); return *p; }

    bool operator==(const Checked_iter& a) const { return p==a.p; }
    bool operator!=(const Checked_iter& a) const { return p!=a.p; }
private:
    void check_end() const { if (p == pc->end()) throw Overflow{}; }
    C* pc {}; // default initialize to nullptr
    typename C::iterator p = pc->begin();
};
```

Очевидно, что это не качество стандартной библиотеки, но это показывает идею:

```
vector<int> v1 {1, 2, 3};           // three elements
vector<int> v2(2);                 // two elements

copy(v1,v2.begin());              // will overflow
copy(v1,Checked_iter{v2});        // will throw
```

Если бы в примере с чтением и сортировкой мы хотели поместить уникальные элементы в новый список, мы могли бы написать:

```
list<Entry> f(vector<Entry>& vec)
{
    list<Entry> res;
    sort(vec.begin(),vec.end());
    unique_copy(vec.begin(),vec.end(),back_inserter(res)); // append to res
```

```

    return res;
}

```

Вызов `back_inserter(res)` создает итератор для `res`, который добавляет элементы в конец контейнера, расширяя контейнер, выделяя для них место. Это избавляет нас от необходимости сначала выделять фиксированный объем памяти, а затем заполнять его. Таким образом, стандартные контейнеры плюс функции `back_inserter()` устраняют необходимость в использовании подверженного ошибкам явного управления памятью в стиле С с помощью `realloc()`. В `list` стандартной библиотеки есть конструктор перемещения (§6.2.2), который делает возврат `res` по значению эффективным (даже для `list` из тысяч элементов).

Когда мы находим код в стиле пары итераторов, такой как `sort(vec.begin(),vec.end())`, утомительным, мы можем использовать различные версии алгоритмов и написать `sort(vec)` (§13.5). Эти две версии эквивалентны. Аналогично, цикл `for` для диапазонов примерно эквивалентен циклу в стиле С, использующему итераторы напрямую:

```

for (auto& x : v) cout<<x;                // write out all elements of v
for (auto p = v.begin(); p!=v.end(); ++p) cout<<*p;    // write out all elements of v

```

В дополнение к тому, что версия `for` для диапазонов проще и менее подвержена ошибкам, она часто также более эффективна.

13.2 Применение итераторов

Для контейнера можно получить несколько итераторов, ссылающихся на полезные элементы; `begin()` и `end()` являются лучшими примерами этого. Кроме того, многие алгоритмы возвращают итераторы. Например, стандартный алгоритм `find` выполняет поиск значения в последовательности и возвращает итератор найденного элемента:

```

bool has_c(const string& s, char c)        // does s contain the character c?
{
    auto p = find(s.begin(),s.end(),c);
    if (p!=s.end())
        return true;
    else
        return false;
}

```

Как и многие алгоритмы поиска в стандартных библиотеках, `find` возвращает `end()` для указания “не найдено”. Эквивалентным, более коротким определением `has_c()` является:

```

bool has_c(const string& s, char c)        // does s contain the character c?
{
    return find(s,c)!=s.end();
}

```

Более интересным упражнением было бы найти местоположение всех вхождений символа в строку. Мы можем вернуть набор вхождений в виде `vector<char*>`. Возврат `vector` эффективен, поскольку `vector` обеспечивает семантику перемещения (§6.2.1). Предполагая, что мы хотели бы изменить найденные местоположения, мы передаем неконстантную строку:

```

vector<string::iterator> find_all(string& s, char c)    // find all occurrences of c
in s
{
    vector<char*> res;
    for (auto p = s.begin(); p!=s.end(); ++p)

```

```

    if (*p==c)
        res.push_back(&*p);
    return res;
}

```

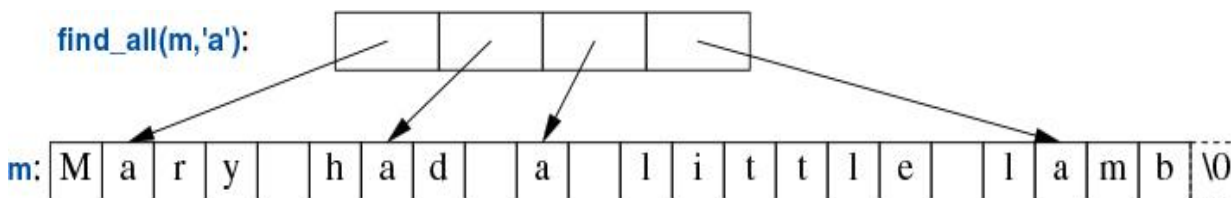
Мы итерируемся по строке, используя обычный цикл, перемещая итератор `p` вперед по одному элементу за раз, при помощи `++`, и просматривая элементы, используя оператор разыменования `*`. Мы можем протестировать `find_all()` следующим образом:

```

void test()
{
    string m {"Mary had a little lamb"};
    for (auto p : find_all(m, 'a' ))
        if (*p!= a )
            cerr << "a bug!\n";
}

```

Этот вызов функции `find_all()` можно было бы графически представить следующим образом:



Итераторы и стандартные алгоритмы работают эквивалентно с каждым стандартным контейнером, для которого их использование имеет смысл. Следовательно, мы могли бы обобщить `find_all()`:

```

template<typename C, typename V>
vector<typename C::iterator> find_all(C& c, V v)    // find all occurrences of v in c
{
    vector<typename C::iterator> res;
    for (auto p = c.begin(); p!=c.end(); ++p)
        if (*p==v)
            res.push_back(p);
    return res;
}

```

Ключевое слово `typename` необходимо для информирования компилятора о том, что итератор `C` должен быть типом, а не значением какого-либо типа, скажем, целым числом `7`.

В качестве альтернативы мы могли бы вернуть вектор обычных указателей на элементы:

```

template<typename C, typename V>
auto find_all(C& c, V v)    // find all occurrences of v in c
{
    vector<range_value_t<C>*> res;
    for (auto& x : c)
        if (x==v)
            res.push_back(&x);
    return res;
}

```

Пока я этим занимался, я также упростил код, используя цикл `for` для диапазонов и `range_value_t` стандартной библиотеки (§16.4.4) для присвоения имен типу элементов. Упрощенная версия `range_value_t` может быть определена следующим образом:

```
template<typename T>
using range_value_type_t = T::value_type;
```

Используя любую версию `find_all()`, мы можем написать:

```
void test()
{
    string m {"Mary had a little lamb"};

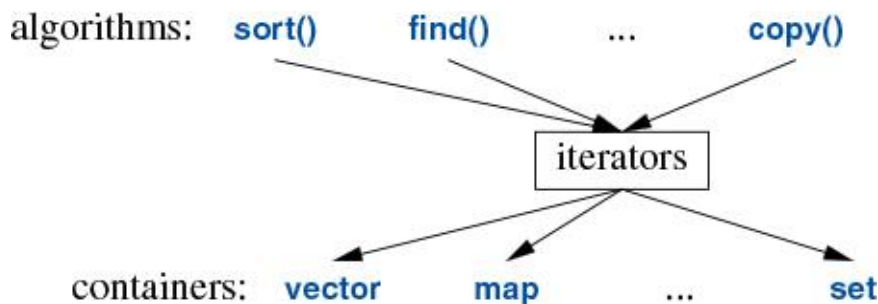
    for (auto p : find_all(m, `a` ))           // p is a string::iterator
        if (*p!= a )
            cerr << "string bug!\n";

    list<int> ld {1, 2, 3, 1, -11, 2};
    for (auto p : find_all(ld,1))              // p is a list<int>::iterator
        if (*p!=1)
            cerr << "list bug!\n";

    vector<string> vs {"red", "blue", "green", "green", "orange", "green"};
    for (auto p : find_all(vs,"red"))          // p is a vector<string>::iterator
        if (*p!="red")
            cerr << "vector bug!\n";

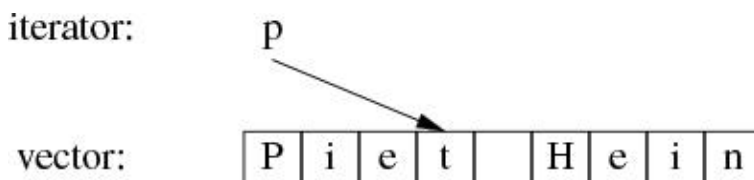
    for (auto p : find_all(vs,"green"))
        *p = "vert";
}
```

Итераторы используются для разделения алгоритмов и контейнеров. Алгоритм оперирует своими данными с помощью итераторов и ничего не знает о контейнере, в котором хранятся элементы. И наоборот, контейнер ничего не знает об алгоритмах, работающих с его элементами; все, что он делает, - это предоставляет итераторы по запросу (например, `begin()` и `end()`). Эта модель разделения между хранилищем данных и алгоритмом обеспечивает очень общее и гибкое программное обеспечение.



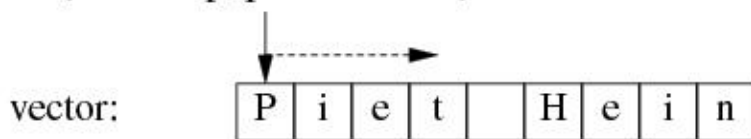
13.3 Типы итераторов

Что такое итераторы на самом деле? Любой конкретный итератор является объектом некоторого типа. Однако существует множество различных типов итераторов – итератор должен содержать информацию, необходимую для выполнения своей работы для определенного типа контейнера. Эти типы итераторов могут быть такими же разными, как контейнеры и специализированные потребности, которым они служат. Например, итератор `vector` может быть обычным указателем, потому что указатель - это вполне разумный способ ссылки на элемент `vector`:



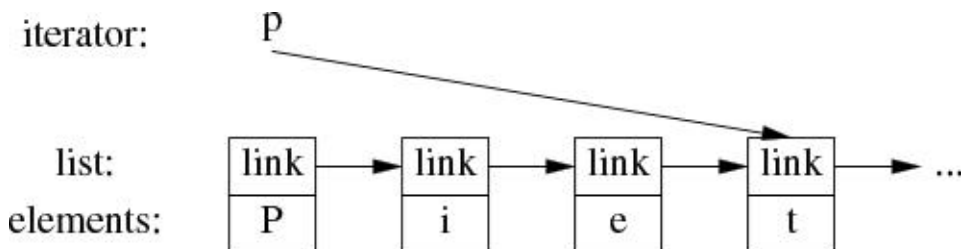
В качестве альтернативы итератор `vector` может быть реализован как указатель на `vector` плюс индекс:

iterator: (start == p, position == 3)



Использование такого итератора позволяет проверять диапазон.

Итератор `list` должен быть чем-то более сложным, чем простой указатель на элемент, потому что элемент `list`, как правило, не знает, где находится следующий элемент этого `list`. Таким образом, итератор `list` может быть указателем на ссылку:



Что является общим для всех итераторов, так это их семантика и названия их операций. Например, применение `++` к любому итератору приводит к созданию итератора, который ссылается на следующий элемент. Аналогично, `*` возвращает элемент, на который ссылается итератор. На самом деле, любой объект, который подчиняется нескольким простым правилам, подобным этим, является итератором. *Итератор* - это общая идея, концепт (§8.2), и различные виды итераторов доступны в качестве `concept` стандартной библиотеки, таких как `forward_iterator` и `random_access_iterator` (§14.5). Кроме того, пользователям редко требуется знать тип конкретного итератора; каждый контейнер “знает” свои типы итераторов и делает их доступными под условными именами `iterator` и `const_iterator`. Например, `list<Entry>::iterator` - это общий тип итератора для `list<Entry>`. Нам редко приходится беспокоиться о деталях определения этого типа.

В некоторых случаях итератор не является типом элемента, поэтому стандартная библиотека предлагает `iterator_t<X>`, который работает везде, где определен итератор `X`.

13.3.1 Поток итераторы

Итераторы - это общая и полезная концепция для работы с последовательностями элементов в контейнерах. Однако контейнеры - не единственное место, где мы находим последовательности элементов. Например, входной поток тоже генерирует последовательность значений, и мы записываем последовательность значений в выходной поток. Следовательно, понятие итераторов может быть с пользой применено к вводу и выводу.

Чтобы создать `ostream_iterator`, нам нужно указать, какой поток будет использоваться, и тип записываемых в него объектов. Например:

```
ostream_iterator<string> oo {cout};           // write strings to cout
```

Результатом присвоения `*oo` является запись присвоенного значения в `cout`.

Например:

```
int main()
{
    *oo = "Hello, ";           // meaning cout<<"Hello, "
    ++oo;
```



```

    *oo = "world!\n";           // meaning cout<<"world!\n"
}

```

Это еще один способ записи канонического сообщения в стандартный вывод. `++oo` выполняется для имитации записи в массив через указатель. Таким образом, мы можем использовать алгоритмы для потоков. Например:

```

vector<string> v{ "Hello", " ", " ", "World!\n" };
copy(v, oo);

```

Аналогично, `istream_iterator` - это то, что позволяет нам обрабатывать входной поток как контейнер, доступный только для чтения. Опять же, мы должны указать используемый поток и тип ожидаемых значений:

```

istream_iterator<string> ii {cin};

```

Итераторы ввода используются парами, представляющими последовательность, поэтому мы должны предоставить `istream_iterator` для указания конца ввода. Это `istream_iterator` по умолчанию:

```

istream_iterator<string> eos {};

```

Как правило, `istream_iterator` и `ostream_iterator` не используются напрямую. Вместо этого они предоставляются в качестве аргументов алгоритмам. Например, мы можем написать простую программу для чтения файла, сортировки прочитанных слов, устранения дубликатов и записи результата в другой файл:

```

int main()
{
    string from, to;
    cin >> from >> to;           // get source and target file names

    ifstream is {from};          // input stream for file "from"
    istream_iterator<string> ii {is}; // input iterator for stream
    istream_iterator<string> eos {}; // input sentinel

    ofstream os {to};             // output stream for file "to"
    ostream_iterator<string> oo {os, "\n"}; // output iterator for stream plus a separator

    vector<string> b {ii, eos};    // b is a vector initialized from input
    sort(b);                      // sort the buffer

    unique_copy(b, oo);           // copy the buffer to output, discard replicated values

    return !is.eof() || !os;      // return error state (§1.2.1, §11.4)
}

```

Я использовал диапазонные версии `sort()` и `unique_copy()`. Я мог бы использовать итераторы напрямую, например, `sort(b.begin(), b.end())`, как это обычно бывает в старом коде.

Пожалуйста, помните, что для использования как традиционной итераторной версии алгоритма стандартной библиотеки, так и его диапазонного аналога, нам нужно либо явно указать вызов версии для диапазона, либо использовать `using` (§9.3.2):

```

copy(v, oo);                     // potentially ambiguous
ranges::copy(v, oo);             // OK
using ranges::copy(v, oo);       // copy(v, oo) OK from here on
copy(v, oo);                     // OK

```

`ifstream` - это `istream`, который может быть прикреплен к файлу (§11.7.2), а `ofstream` - это `ostream`, который может быть прикреплен к файлу. Вторым аргументом `ostream_iterator` используется для разграничения выходных значений.

На самом деле, эта программа длиннее, чем должна быть. Мы считываем строки в `vector`, затем `sort()` их, а затем записываем, устраняя дубликаты. Более элегантным решением является вообще не хранить дубликаты. Это можно сделать, сохранив `string` в `set`, который не содержит дубликатов и поддерживает порядок своих элементов (§12.5). Таким образом, мы могли бы заменить две строки, использующие `vector`, на одну, использующую `set`, и заменить `unique_copy()` более простой `copy()`:

```
set<string> b {ii,eos};           // collect strings from input
copy(b,oo);                      // copy buffer to output
```

Мы использовали имена `ii`, `eos` и `oo` только один раз, чтобы еще больше уменьшить размер программы:

```
int main()
{
    string from, to;
    cin >> from >> to;           // get source and target file names

    ifstream is {from};          // input stream for file "from"
    ofstream os {to};            // output stream for file "to"

    set<string> b {istream_iterator<string>{is},istream_iterator<string>{}}; // read
input
    copy(b,ostream_iterator<string>{os,"\\n"}); // copy to output

    return !is.eof() || !os;     // return error state (§1.2.1, §11.4)
}
```

Улучшит ли это последнее упрощение читабельность или нет, зависит от вкуса и опыта.

13.4 Использование предикатов

В приведённых до настоящего времени примерах алгоритмы просто “встроены” в действие, которое необходимо выполнить для каждого элемента последовательности. Однако мы часто хотим сделать это действие параметром алгоритма. Например, алгоритм `find` (§13.2, §13.5) предоставляет удобный способ поиска определенного значения. Более общий вариант ищет элемент, удовлетворяющий указанному требованию - *предикат*. Например, мы могли бы захотеть выполнить поиск в `map` первого значения, большего чем 42. `map` позволяет нам получать доступ к ее элементам в виде последовательности пар (*ключ, значение*), поэтому мы можем искать `pair<const string,int>` в последовательности `map<string,int>`, где значение `int` больше 42:

```
void f(map<string,int>& m)
{
    auto p = find_if(m, Greater_than{42});
    // ...
}
```

Здесь `Greater_than` - это функциональный объект (§7.3.2), содержащий значение (42) для сравнения с записью карты типа `pair<string,int>`:

```
struct Greater_than {
    int val;
    Greater_than(int v) : val{v} { }
```

```
bool operator()(const pair<string,int>& r) const { return r.second>val; }
};
```

Альтернативно и эквивалентно, мы могли бы использовать лямбда-выражение (§7.3.2):

```
auto p = find_if(m, [](const auto& r) { return r.second>42; });
```

Предикат не должен изменять элементы, к которым он применяется.

13.5 Обзор алгоритмов

Общее определение алгоритма - это “конечный набор правил, который задает последовательность операций для решения определенного набора задач [и] обладает пятью важными характеристиками: Конечностью ... Определенностью ... Вводом ... Выводом ... Эффективностью” [Knuth,1968,§1.1]. В контексте стандартной библиотеки C++ алгоритм - это шаблон функции, работающий с последовательностями элементов.

Стандартная библиотека предоставляет несколько десятков алгоритмов. Алгоритмы определены в пространстве имен `std` и представлены в заголовках `<algorithm>` и `<numeric>`. Все эти алгоритмы стандартной библиотеки принимают последовательности в качестве входных данных. Полуоткрытая последовательность от `b` до `e` называется `[b:e)`. Вот несколько примеров:

Избранные стандартные алгоритмы <code><algorithm></code>	
<code>f=for_each(b,e,f)</code>	Для каждого элемента <code>x</code> в <code>[b:e)</code> выполнить <code>f(x)</code>
<code>p=find(b,e,x)</code>	<code>p</code> это первый элемент в <code>[b:e)</code> такой что <code>*p==x</code>
<code>p=find_if(b,e,f)</code>	<code>p</code> это первый элемент в <code>[b:e)</code> такой что <code>f(*p)==true</code>
<code>n=count(b,e,x)</code>	<code>n</code> количество элементов <code>*q</code> в <code>[b:e)</code> такое что <code>*q==x</code>
<code>n=count_if(b,e,f)</code>	<code>n</code> количество элементов <code>*q</code> в <code>[b:e)</code> такое что <code>f(*q)==true</code>
<code>replace(b,e,v,v2)</code>	Заменить элементы <code>*q</code> в <code>[b:e)</code> такие что <code>*q==v</code> элементами <code>v2</code>
<code>replace_if(b,e,f,v2)</code>	Заменить элементы <code>*q</code> в <code>[b:e)</code> такие что <code>f(*q)==true</code> элементами <code>v2</code>
<code>p=copy(b,e,out)</code>	Копировать <code>[b:e)</code> в <code>[out:p)</code>
<code>p=copy_if(b,e,out,f)</code>	Копировать элементы <code>*q</code> из <code>[b:e)</code> такие что <code>f(*q)==true</code> в <code>[out:p)</code>
<code>p=move(b,e,out)</code>	Переместить <code>[b:e)</code> в <code>[out:p)</code>
<code>p=unique_copy(b,e,out)</code>	Копировать <code>[b:e)</code> в <code>[out:p)</code> ; не копирует последующие дубликаты элем.
<code>sort(b,e)</code>	Сортировать элементы <code>[b:e)</code> используя <code><</code> в качестве условия сортировки
<code>sort(b,e,f)</code>	Сортировать элементы <code>[b:e)</code> используя <code>f</code> в качестве условия сортировки
<code>(p1,p2)=equal_range(b,e,v)</code>	<code>[p1:p2)</code> поддиапазон отсортированной последовательности <code>[b:e)</code> со значением <code>v</code> ; по сути, это бинарный поиск для <code>v</code>
<code>p=merge(b,e,b2,e2,out)</code>	Объединить две отсортированные последовательности <code>[b:e)</code> и <code>[b2:e2)</code> в <code>[out:p)</code>
<code>p=merge(b,e,b2,e2,out,f)</code>	Объединить две отсортированные последовательности <code>[b:e)</code> и <code>[b2:e2)</code> в <code>[out:p)</code> используя <code>f</code> как компаратор

Для каждого алгоритма, принимающего диапазон `[b:e)`, `<ranges>` предлагает версию, которая принимает диапазон. Пожалуйста, помните (§9.3.2), что для использования как

традиционной итераторной версии алгоритма стандартной библиотеки, так и его аналога с диапазонами, вам необходимо либо явно квалифицировать вызов, либо использовать `using`.

Эти и многие другие алгоритмы (например, §17.3) могут быть применены к элементам контейнеров, `string` и встроеным массивам.

Некоторые алгоритмы, такие как `replace()` и `sort()`, изменяют значения элементов, но ни один алгоритм не добавляет или не удаляет элементы контейнера. Причина в том, что последовательность не идентифицирует контейнер, содержащий элементы последовательности. Чтобы добавлять или удалять элементы, вам нужно что-то, что знает о контейнере (например, `back_inserter`; §13.1) или непосредственно ссылается на сам контейнер (например, `push_back()` или `erase()`; §12.2).

Лямбды очень распространены в качестве операций, передаваемых в качестве аргументов. Например:

```
vector<int> v = {0,1,2,3,4,5};  
for_each(v, [](int& x){ x=x*x; });           // v=={0,1,4,9,16,25}  
for_each(v.begin(),v.begin()+3, [](int& x){ x=sqrt(x); }); // v=={0,1,2,9,16,25}
```

Алгоритмы стандартной библиотеки, как правило, более тщательно разработаны, специфицированы и реализованы, чем обычный цикл, созданный вручную. Знайте их и используйте вместо кода, написанного на "голом" языке.

13.6 Параллельные алгоритмы

Когда одна и та же задача должна быть выполнена со многими элементами данных, мы можем выполнять ее параллельно для каждого элемента данных при условии, что вычисления для разных элементов данных независимы:

- *параллельное выполнение*: задачи выполняются в нескольких потоках (часто на нескольких процессорных ядрах)
- *векторизованное выполнение*: задачи выполняются в одном потоке с использованием векторизации, также известной как SIMD ("Одна инструкция, множество данных").

Стандартная библиотека предлагает поддержку и того, и другого, и мы можем точно указать необходимость последовательного выполнения; в `<execution>` в пространстве имён `execution` мы находим:

- `seq`: последовательное выполнение
- `par`: параллельное выполнение (если возможно)
- `unseq`: непоследовательное (векторизованное) выполнение (если возможно)
- `par_unseq`: параллельное и/или непоследовательное (векторизованное) выполнение (если это возможно). Рассмотрим `std::sort()`:

```
sort(v.begin(),v.end());           // sequential  
sort(seq,v.begin(),v.end());       // sequential (same as the default)  
sort(par,v.begin(),v.end());       // parallel  
sort(par_unseq,v.begin(),v.end()); // parallel and/or vectorized
```

Стоит ли распараллеливать и/или векторизовать, зависит от алгоритма, количества элементов в последовательности, аппаратного обеспечения и использования этого оборудования программами, запущенными на нём. Следовательно, *индикаторы политики исполнения* - это всего лишь подсказки. Компилятор и/или планировщик времени

выполнения будут решать, какой объем параллелизма использовать. Все это нетривиально, и здесь очень важно правило, запрещающее делать заявления об эффективности без проведения измерений.

К сожалению, расширенные версии параллельных алгоритмов еще не включены в стандарт, но если они нам понадобятся, их легко определить:

```
void sort(auto pol, random_access_range auto& r)
{
    sort(pol,r.begin(),r.end());
}
```

Большинство алгоритмов стандартной библиотеки, включая все из таблицы в §13.5, за исключением `equal_range`, могут запросить распараллеливание и векторизацию с использованием `par` и `par_unseq`, как для `sort()`. Почему не `equal_range()`? Потому что до сих пор никто не придумал для этого стоящего параллельного алгоритма.

Многие параллельные алгоритмы используются в основном для обработки числовых данных; см. §17.3.1.

Запрашивая параллельное выполнение, обязательно избегайте состояния гонки за данные (§18.2) и взаимоблокировок (§18.3).

13.7 Советы

- [1] Алгоритм STL работает с одной или несколькими последовательностями; §13.1.
- [2] Входная последовательность является полуоткрытой и определяется парой итераторов; §13.1.
- [3] Вы можете определить свои собственные итераторы для удовлетворения особых потребностей; §13.1.
- [4] К потокам ввода-вывода можно применить множество алгоритмов; §13.3.1.
- [5] Алгоритм поиска обычно возвращает конец входной последовательности, чтобы указать “не найдено”; §13.2.
- [6] Алгоритмы напрямую не добавляют и не удаляют элементы из последовательностей переданных в качестве аргументов; §13.2, §13.5.
- [7] При написании цикла подумайте, можно ли его выразить в виде общего алгоритма; §13.2.
- [8] Используйте псевдонимы `using` для очистки беспорядочных обозначений; §13.2.
- [9] Используйте предикаты и другие функциональные объекты, чтобы придать стандартным алгоритмам более широкий диапазон применений; §13.4, §13.5.
- [10] Предикат не должен изменять свой аргумент; §13.4.
- [11] Знайте свои алгоритмы из стандартной библиотеки и предпочитайте их циклам, созданным вручную; §13.5.

Диапазоны

*Самые веские аргументы ничего
не доказывают до тех пор, пока
выводы не подтверждаются
опытом.
– Roger Bacon*

- [Введение](#)
- [Представления](#)
- [Генераторы](#)
- [Конвейеры](#)
- [Обзор концептов](#)

[Концепты типов](#); [Концепты итераторов](#); [Концепты диапазонов](#)

- [Советы](#)

14.1 Введение

Стандартная библиотека предлагает алгоритмы как ограниченные при помощи концептов ([глава 8](#)), так и без ограничений (для совместимости). Ограниченные (концептами) версии находятся в `<ranges>` в пространстве имён `ranges`. Естественно, я предпочитаю версии, использующие концепты. `range` - это обобщение последовательностей из C++98, определённых парами `{begin(), end()}`; диапазон определяет, что требуется для того, чтобы быть последовательностью элементов. `range` может быть определен с помощью

- Пары итераторов `{begin, end}`
- Пары `{begin, n}`, где `begin` это итератор, а `n` это количество элементов
- Пары `{begin, pred}`, где `begin` это итератор, а `pred` это предикат; если `pred(p)` возвращает `true` для итератора `p`, когда мы достигли конца диапазона. Это позволяет нам иметь бесконечные диапазоны, которые генерируются “на лету” (§14.3).

Этот концепт `range` позволяет нам написать `sort(v)`, а не `sort(v.begin(), v.end())`, как мы привыкли использовать STL с 1994 года. Мы можем сделать то же самое для наших собственных алгоритмов:

```
template<forward_range R>
    requires sortable<iterator_t<R>>
void my_sort(R& r) // modern, concept-constrained version of
my_sort
{
    return my_sort(r.begin(), end()); // use the 1994-style sort
}
```

Диапазоны позволяют нам более точно выразить примерно 99% распространенных применений алгоритмов. В дополнение к удобству записи, диапазоны предлагают некоторые возможности для оптимизации и устраняют класс “глупых ошибок”, таких как `sort(v1.begin(), v2.end())` и `sort(v.end(), v.begin())`. Да, такие ошибки были замечены “в дикой природе”.

Естественно, существуют различные типы диапазонов, соответствующие различным типам итераторов. В частности, `input_range`, `forward_range`, `bidirectional_range`, `random_access_range` и `contiguous_range` представлены как концепты (§14.5).

14.2 Представления

Представление - это способ выразить диапазон. Например:

```
void user(forward_range auto& r)
{
    filter_view v {r, [](int x) { return x%2; } }; // view (only) odd numbers from r

    cout << "odd numbers: "
    for (int x : v)
        cout << x << ' ';
}
```

При чтении из `filter_view` мы считываем данные из его диапазона. Если считанное значение соответствует предикату, оно возвращается; в противном случае `filter_view` повторяет попытку со следующим элементом из диапазона.

Многие диапазоны бесконечны. Кроме того, нам часто нужно всего несколько значений. Следовательно, существуют представления для получения только нескольких значений из диапазона:

```
void user(forward_range auto& r)
{
    filter_view v{r, [](int x) { return x%2; } }; // view (only) odd numbers in r
    take_view tv {v, 100 }; // view at most 100 element
    from v

    cout << "odd numbers: "
    for (int x : tv)
        cout << x << ' ';
}
```

Мы можем избежать присвоения имени `take_view`, используя его напрямую:

```
for (int x : take_view{v, 3})
    cout << x << ' ';
```

Аналогично для `filter_view`:


```
for (int x : take_view{ filter_view { r, [](int x) { return x % 2; } }, 3 })
    cout << x << ' ';
```

Такое вложение представлений может быстро стать нечитаемым, поэтому есть альтернатива: конвейеры (§14.4).

Стандартная библиотека предлагает множество представлений, также известных как *адаптеры диапазона*:

Представления стандартной библиотеки (адаптеры диапазона) <ranges>	
v это view; r это range; p это предикат; n это целое число	
v=all_view{r}	v это все элементы из r
v=filter_view{r,p}	v это элементы из r которые соответствуют p
v=transform_view{r,f}	v это результат вызова f для каждого элемента из r
v=take_view{r,n}	v это не более n элементов из r
v=take_while_view{r,p}	v это элементы из r пока не выполнится p
v=drop_view{r,n}	v это элементы из r начиная с n+1 элемента
v=drop_while_view{r,p}	v это элементы из r начиная с первого элемента, который не соответствует p
v=join_view{r}	v развернутая версия r; элементы r должны быть диапазонами
v=split_view(r,d)	v это диапазон из поддиапазонов r определяемый разделителем d; d должен быть элементом или диапазоном
v=common_view(r)	v это r записанный парой (begin:end)
v=reverse_view{r}	v это элементы из r в обратном порядке; r должен иметь двунаправленный доступ
v=views::elements<n>(r)	v это диапазон из n-ных элементов из tuple, являющихся элементами r
v=keys_view{r}	v это диапазон из первых элементов pair, являющихся элементами r
v=values_view{r}	v это диапазон из вторых элементов pair, являющихся элементами r
v=ref_view{r}	v это диапазон из ссылок на элементы r

Представление предлагает интерфейс, очень похожий на интерфейс диапазона, поэтому в большинстве случаев мы можем использовать представление везде, где мы можем использовать диапазон, и таким же образом. Ключевое отличие заключается в том, что представление не владеет своими элементами; оно не несет ответственности за удаление элементов из своего базового диапазона - это ответственность диапазона. С другой стороны, представление не должно выходить за рамки своего диапазона:

```
auto bad()
{
    vector v = {1, 2, 3, 4};
    return filter_view{v,odd};           // v will be destroyed before the view
}
```

Предполагается, что представления дешевы для копирования, поэтому мы передаем их по значению.

Я использовал простые стандартные типы, чтобы примеры были тривиальными, но, конечно, у нас могут быть представления наших собственных пользовательских типов. Например:

```

struct Reading {
    int location {};
    int temperature {};
    int humidity {};
    int air_pressure {};
    //...
};

int average_temp(vector<Reading> readings)
{
    if (readings.size()==0) throw No_readings{};
    double s = 0;
    for (int x: views::elements<1>(readings))    // look at just the temperatures
        s += x;
    return s/readings.size();
}

```

14.3 Генераторы

Часто диапазон необходимо генерировать "на лету". Стандартная библиотека предоставляет для этого несколько простых *генераторов* (они же *фабрики*):

Фабрики диапазонов <ranges> v это представление; x это элемент типа T; is это istream	
v=empty_view<T>{}	v это пустой диапазон типа T elements (had it had any)
v=single_view{x}	v это диапазон из одного элемента x
v=iota_view{x}	v это бесконечный диапазон элементов: x , x+1 , x+2 , ... инкремент выполняется с помощью ++
v=iota_view{x,y}	v это диапазон из n элементов: x , x+1 , ..., y-1 инкремент выполняется с помощью ++
v=istream_view<T>{is}	v это диапазон полученный вызовом >> для T из is

iota_view полезны для генерации простых последовательностей. Например:

```

for (int x : iota_view(42,52))    // 42 43 44 45 46 47 48 49 50 51
    cout << x << ' ';

```

istream_view дает нам простой способ использования **istream** в циклах range-for:

```

for (auto x : istream_view<complex<double>>(cin) )
    cout << x << '\n';

```

Как и другие представления, **istream_view** может быть составлен из других представлений:

```

auto cplx = istream_view<complex<double>>(cin);

for (auto x : transform_view(cplx, [](auto z){ return z*z;}))
    cout << x << '\n';

```

При вводе **1 2 3** получается **1 4 9**.

14.4 Конвейеры

Для каждого представления из стандартной библиотеки (§14.2) есть функция, которая создает фильтр; то есть объект, который может использоваться в качестве аргумента оператору фильтра **|**. Например, **filter()** возвращает **filter_view**. Это позволяет нам

комбинировать фильтры в конвейер, а не представлять их в виде набора вложенных вызовов функций.

```
void user(forward_range auto& r)
{
    auto odd = [](int x) { return x % 2; };

    for (int x : r | views::filter(odd) | views::take(3))
        cout << x << ' ';
}
```

Диапазон входных данных **2 4 6 8 20** дает значение **1 2 3**.

Конвейерный стиль (использующий оператор конвейера Unix `|`) широко считается более удобочитаемым, чем вызовы вложенных функций. Конвейер работает слева направо; то есть `f | g` результат `f` передается в `g`, поэтому `r | f | g` означает `(g_filter (f_filter (r)))`. Начальное значение `r` должно быть диапазоном или генератором.

Эти функции фильтрации находятся в пространстве имён `ranges::views`:

```
void user(forward_range auto& r)
{
    for (int x : r | views::filter([](int x) { return x % 2; } ) | views::take(3) )
        cout << x << ' ';
}
```

Я нахожу, что использование `views::` явно делает код вполне читабельным, но, конечно, мы можем еще больше улучшить код:

```
void user(forward_range auto& r)
{
    using namespace views;

    auto odd = [](int x) { return x % 2; };

    for (int x : r | filter(odd) | take(3) )
        cout << x << ' ';
}
```

Реализация представлений и конвейеров включает в себя довольно сложное мета-программирование шаблонов, поэтому, если вы беспокоитесь о производительности, обязательно измерьте, обеспечивает ли ваша реализация то, что вам нужно. Если нет, то всегда есть обычный обходной путь:

```
void user(forward_range auto& r)
{
    int count = 0;
    for (int x : r)
        if (x % 2) {
            cout << x << ' ';
            if (++count == 3) return;
        }
}
```

Однако здесь логика происходящего затуманена.

14.5 Обзор концептов

Стандартная библиотека предлагает множество полезных концептов:

- Концепты определяющие свойства типов (§14.5.1)
- Концепты определяющие итераторы (§14.5.2)

- Концепты определяющие диапазоны (§14.5.3)

14.5.1 Концепты типов

Концепты, связанные со свойствами типов и отношениями между типами, отражают разнообразие типов. Эти концепты помогают упростить большинство шаблонов.

Основные языковые концепты <concepts>	
T и U это типы	
<code>same_as<T,U></code>	T такой же как и U
<code>derived_from<T,U></code>	T является производным от U
<code>convertible_to<T,U></code>	T может быть преобразован в U
<code>common_reference_with<T,U></code>	T и U совместно используют общий ссылочный тип
<code>common_with<T,U></code>	T и U имеют общий тип
<code>integral<T></code>	T является целочисленным типом
<code>signed_integral<T></code>	T является целочисленным типом со знаком
<code>unsigned_integral<T></code>	T является целочисленным типом без знака
<code>floating_point<T></code>	T является типом число с плавающей точкой
<code>assignable_from<T,U></code>	U может быть присвоен T
<code>swappable_with<T,U></code>	T может обмениваться значениями с U
<code>swappable<T></code>	<code>swappable_with<T,T></code>

Многие алгоритмы должны работать с комбинациями связанных типов, например, выражениями со смесью `int` и `double`. Мы используем `common_with`, чтобы сказать, является ли такое сочетание математически обоснованным. Если `common_with<X,Y>` возвращает `true`, мы можем использовать `common_type_t<X,Y>` для сравнения X с Y, сначала преобразовав оба в `common_type_t<X,Y>`. Например:

```
common_type<string, const char*> s1 = some_fct()
common_type<string, const char*> s2 = some_other_fct();

if (s1<s2) {
    // ...
}
```

Чтобы указать общий тип для пары типов, мы специализируем `common_type_t`, используемый в определении `common`. Например:

```
using common_type_t<Bigint,long> = Bigint;    // for a suitable definition of Bigint
```

К счастью, нам не нужно определять специализацию `common_type_t`, если только мы не хотим использовать операции со смесями типов, для которых в библиотеке (пока) нет подходящих определений.

На концепты, связанные со сравнением, сильно повлиял [Stepanov,2009].

Концепты для сравнений <concepts>	
<code>equality_comparable_with<T,U></code>	T и U могут проверяться на равенство при помощи <code>==</code>
<code>equality_comparable<T></code>	<code>equality_comparablewith<T,T></code>
<code>totally_ordered_with<T,U></code>	T и U могут сравниваться при помощи <code><</code> , <code><=</code> , <code>></code> и <code>>=</code> возвращает общий порядок
<code>totally_ordered<T></code>	<code>strict_totally_ordered_with<T,T></code>

<code>three_way_comparable_with<T,U></code>	<code>T</code> и <code>U</code> могут сравниваться при помощи <code><=></code> возвращает стабильный результат
<code>three_way_comparable<T></code>	<code>three_way_comparable_with<T,T></code>

Использование как `equality_comparable_with`, так и `equality_comparable` показывает (пока) упущенную возможность перегрузки концептов.

Любопытно, что стандартного концепта `boolean` не существует. Мне это часто нужно, так что вот версия:

```
template<typename B>
concept Boolean =
    requires(B x, B y) {
        { x = true };
        { x = false };
        { x = (x == y) };
        { x = (x != y) };
        { x = !x };
        { x = (x = y) };
    };
};
```

При написании шаблонов нам часто необходимо классифицировать типы.

Концепты объектов <concepts>	
<code>destructible<T></code>	<code>T</code> может быть уничтожен, и его адрес можно записать с помощью унарного <code>&</code>
<code>constructible_from<T,Args></code>	<code>T</code> может быть сконструирован из списка аргументов типа <code>Args</code>
<code>default_initializable<T></code>	<code>T</code> имеет конструктор по умолчанию
<code>move_constructible<T></code>	<code>T</code> имеет move-конструктор
<code>copy_constructible<T></code>	<code>T</code> имеет copy-конструктор и move-конструктор
<code>movable<T></code>	<code>move_constructible<T></code> , <code>assignable<T&,T></code> и <code>swappable<T></code>
<code>copyable<T></code>	<code>copy_constructible<T></code> , <code>moveable<T></code> и <code>assignable<T, const T&></code>
<code>semiregular<T></code>	<code>copyable<T></code> и <code>default_constructible<T></code>
<code>regular<T></code>	<code>semiregular<T></code> и <code>equality_comparable<T></code>

Идеальный вариант для типов это `regular`. Тип соответствующий `regular` работает примерно так же, как `int`, и упрощает большую часть наших размышлений о том, как использовать этот тип (§8.2). Отсутствие по умолчанию `==` для классов означает, что большинство классов относятся к `semiregular`, хотя большинство из них могли бы и должны быть `regular`.

Всякий раз, когда мы передаем операцию в качестве аргумента ограниченного шаблона, нам нужно указать, как она может быть вызвана, а иногда и какие предположения мы делаем об их семантике.

Концепты вызовов <concepts>	
<code>invocable<F, Args></code>	<code>F</code> может быть вызван со списком аргументов типа <code>Args</code>
<code>regular_invocable<F, Args></code>	<code>invocable<F, Args></code> и сохраняется ли равенство
<code>predicate<F, Args></code>	<code>regular_invocable<F, Args></code> возвращающий <code>bool</code>
<code>relation<F, T, U></code>	<code>predicate<F, T, U></code>
<code>equivalence_relation<F, T, U></code>	<code>relation<F, T, U></code> это обеспечивает отношение эквивалентности
<code>strict_weak_order<F, T, U></code>	<code>relation<F, T, U></code> это обеспечивает строгое слабое упорядочение

Функция `f()` сохраняет равенство, если `x==y` подразумевает, что `f(x)==f(y)`. `invocable` и `regular_invocable` отличаются только семантически. Мы не можем (в настоящее время) представить это в коде, поэтому имена просто выражают наши намерения.

Аналогично, `relation` и `equivalence_relation` отличаются только семантически. Отношение эквивалентности является рефлексивным, симметричным и транзитивным.

`relation` и `strict_weak_order` отличаются только семантически. Строгий слабый порядок - это то, что стандартная библиотека обычно предполагает для сравнений, таких как `<`.

14.5.2 Концепты итераторов

Традиционные стандартные алгоритмы получают доступ к своим данным через итераторы, поэтому нам нужны концепты для классификации свойств типов итераторов.

Концепты итераторов <code><iterators></code>	
<code>input_or_output_iterator<I></code>	<code>I</code> может быть инкрементирован (<code>++</code>) и разыменован (<code>*</code>)
<code>sentinel_for<S, I></code>	<code>S</code> является стражем для типа <code>I</code> ; это значит, <code>S</code> это предикат для значения типа <code>I</code>
<code>sized_sentinel_for<S, I></code>	Страж <code>S</code> где оператор <code>-</code> может быть применён к <code>I</code>
<code>input_iterator<I></code>	<code>I</code> это итератор ввода; разыменование <code>*</code> может быть использовано только для чтения
<code>output_iterator<I></code>	<code>I</code> это итератор вывода; разыменование <code>*</code> может быть использовано только для записи
<code>forward_iterator<I></code>	<code>I</code> однонаправленный итератор, поддерживающий многопроходность и <code>==</code>
<code>bidirectional_iterator<I></code>	<code>forward_iterator<I></code> поддерживающий <code>--</code>
<code>random_access_iterator<I></code>	<code>bidirectional_iterator<I></code> поддерживающий <code>+</code> , <code>-</code> , <code>+=</code> , <code>-=</code> и <code>[]</code>
<code>contiguous_iterator<I></code>	<code>random_access_iterator<I></code> для элементов в непрерывной памяти
<code>permutable<I></code>	<code>forward_iterator<I></code> с поддержкой перемещения и замены
<code>mergeable<I1, I2, R, O></code>	Может ли объединять отсортированные последовательности, определенные с помощью <code>I1</code> и <code>I2</code> в <code>O</code> используя <code>relation<R></code> ?
<code>sortable<I></code>	Может ли сортировать последовательности, определенные с помощью <code>I</code> используя <code>less</code> ?
<code>sortable<I, R></code>	Может ли сортировать последовательности, определенные с помощью <code>I</code> используя <code>relation<R></code> ?

`mergeable` и `sortable` упрощены по сравнению с их определением в C++20.

Различные виды (типы) итераторов используются для выбора наилучшего алгоритма для заданного набора аргументов; см. §8.2.2 и §16.4.1. Пример `input_iterator` приведен в §13.3.1.

Основная идея стража заключается в том, что мы можем выполнять итерацию по диапазону, начиная с итератора, до тех пор, пока предикат не станет истинным для элемента. Таким образом, итератор `p` и страж `s` определяют диапазон `[p:s(*p))`. Например, мы могли бы определить предикат для стража для обхода строки в стиле C, используя

указатель в качестве итератора. К сожалению, для этого требуется некоторый излишний код, потому что идея состоит в том, чтобы представить предикат как нечто, что нельзя спутать с обычным итератором, но что вы можете сравнить с итератором, используемым для перебора элементов диапазона:

```
template<class Iter>
class Sentinel {
public:
    Sentinel(int ee) : end(ee) { }
    Sentinel() :end(0) {}           // Concept sentinel_for requires a default constructor

    friend bool operator==(const Iter& p, Sentinel s) { return (*p == s.end); }
    friend bool operator!=(const Iter& p, Sentinel s) { return !(p == s); }
private:
    iter_value_t<const char*> end;    // the sentinel value
};
```

Объявление **friend** позволяет нам определить бинарные операторы **==** и **!=** для сравнения итератора со стражем в рамках класса.

Мы можем проверить, что этот **Sentinel** соответствует требованиям **sentinel_for** для **const char***:

```
static_assert(sentinel_for<Sentinel<const char*>, const char*>); //
check the Sentinel for C-style strings
```

Наконец, мы можем написать довольно своеобразную версию программы “Привет, мир!”:

```
const char aa[] = "Hello, World!\nBye for now\n";

ranges::for_each(aa, Sentinel<const char*>('\n'), [](const char x) { cout << x; });
```

Да, это действительно выведет **Hello, World!** без перехода на новую строку.

14.5.3 Концепты диапазонов

Концепты диапазонов определяют свойства диапазонов.

Концепты диапазонов <ranges>	
range<R>	R это диапазон с начальным итератором и стражем
sized_range<R>	R это диапазон, который всегда знает свой размер
view<R>	R это диапазон с постоянным временем копирования, перемещения и присвоения
common_range<R>	R это диапазон с идентичными типами итератора и стража
input_range<R>	R это диапазон, тип итератора которого удовлетворяет концепту input_iterator
output_range<R>	R это диапазон, тип итератора которого удовлетворяет концепту output_iterator
forward_range<R>	R это диапазон, тип итератора которого удовлетворяет концепту forward_iterator
bidirectional_range<R>	R это диапазон, тип итератора которого удовлетворяет концепту bidirectional_iterator
random_access_range<R>	R это диапазон, тип итератора которого удовлетворяет концепту random_access_iterator

<code>contiguous_range<R></code> <code>R</code> это диапазон, тип итератора которого удовлетворяет концепту <code>contiguous_iterator</code>
--

В `<ranges>` есть еще больше концептов, но этот набор - хорошее начало. Основное использование этих концептов заключается в том, чтобы разрешить перегрузку реализаций на основе свойств типа их входных данных (§8.2.2).

14.6 Советы

- [1] Когда стиль парных итераторов становится утомительным, используйте диапазонные алгоритмы; §13.1; §14.1.
- [2] При использовании алгоритма диапазона не забудьте явно указать его название; §13.3.1.
- [3] Конвейеры операций над диапазоном могут быть выражены с помощью `view`, `generator` и `filter`; §14.2, §14.3, §14.4.
- [4] Чтобы завершить диапазон предикатом, вам нужно определить стража; §14.5.
- [5] Используя `static_assert`, мы можем проверить, соответствует ли конкретный тип требованиям концепта; §8.2.4.
- [6] Если вам нужен алгоритм для диапазона, а в стандарте его нет, просто напишите свой собственный; §13.6.
- [7] Идеальным вариантом для типов является `regular`; §14.5.
- [8] Предпочитайте концепты стандартных библиотек там, где они применимы; §14.5.
- [9] Запрашивая параллельное выполнение, обязательно избегайте состояния гонки за данные (§18.2) и взаимных блокировок (§18.3); §13.6.

Умные указатели и контейнеры

*Образование - это то, что, когда
и зачем нужно делать.*

*Тренировка - это то, как это сде-
лать.*

– Richard Hamming

- [Введение](#)
- [Указатели](#)

`unique_ptr` и `shared_ptr`; `span`

- [Контейнеры](#)

`array`; `bitset`; `pair`; `tuple`

- [Альтернативы](#)

`variant`; `optional`; `any`

- [Советы](#)

15.1 Введение

C++ предлагает простые встроенные низкоуровневые типы для хранения данных и ссылок на них: объекты и массивы содержат данные; указатели и массивы ссылаются на такие данные. Однако нам необходимо поддерживать как более специализированные, так и более общие способы хранения и использования данных. Например, контейнеры стандартной библиотеки ([глава 12](#)) и итераторы (§13.3) предназначены для поддержки основных алгоритмов.

Основная общность абстракций контейнера и указателя заключается в том, что их правильное и эффективное использование требует инкапсуляции данных вместе с набором функций для доступа к ним и управления ими. Например, указатели являются очень общими и эффективными абстракциями машинных адресов, но их правильное использование для представления владения ресурсами оказалось чрезмерно сложным. Итак, стандартная библиотека предлагает умные указатели для управления ресурсами; то есть классы, которые инкапсулируют указатели и предоставляют операции, упрощающие их правильное использование.

Эти абстракции стандартной библиотеки инкапсулируют встроенные языковые типы и их бысродействие и занимаемое пространство так же хорошо, как и при правильном использовании этих типов.

В этих типах нет ничего “волшебного”. Мы можем разрабатывать и внедрять наши собственные “умные указатели” и специализированные контейнеры по мере необходимости, используя те же методы, которые используются для стандартных библиотек.

15.2 Указатели

Общее понятие *указателя* - это нечто, что позволяет нам ссылаться на объект и получать к нему доступ в соответствии с его типом. Встроенный указатель, такой как `int*`, является примером, но их гораздо больше.

Указатели	
<code>T*</code>	Встроенный тип указателя: указывает на объект типа <code>T</code> или к непрерывно-аллоцированной последовательности элементов типа <code>T</code>
<code>T&</code>	Встроенный ссылочный тип: ссылается на объект типа <code>T</code> ; указатель с неявным разыменованием (§1.7)
<code>unique_ptr<T></code>	Владеющий указатель на <code>T</code>
<code>shared_ptr<T></code>	Указатель на объект типа <code>T</code> ; право собственности распределяется между всеми <code>shared_ptr</code> на этот <code>T</code>
<code>weak_ptr<T></code>	Указатель на объект, принадлежащий <code>shared_ptr</code> ; должен быть преобразован в <code>shared_ptr</code> для доступа к объекту
<code>span<T></code>	Указатель на непрерывную последовательность <code>T</code> (§15.2.2)
<code>string_view<T></code>	Указатель на константную <code>const</code> подстроку (§10.3)
<code>X_iterator<C></code>	Последовательность элементов из <code>C</code> ; Символ <code>X</code> в названии указывает на тип итератора (§13.3)

На объект может быть более одного указателя. Указатель-владелец - это тот, который отвечает за конечное удаление объекта, на который он ссылается. Указатель, *не являющийся владельцем* (например, `T*` или `span`), может *повиснуть*; то есть указывать на местоположение, где объект был `deleted` или вышел за пределы области видимости.

Чтение или запись с помощью повисшего указателя - один из самых неприятных видов ошибок. Результат этого технически не определен. На практике это часто означает доступ к объекту, который случайно находится в этом местоположении. Тогда чтение означает получение произвольного значения, а запись перезаписывает несвязанную структуру данных. Лучшее, на что мы можем надеяться, - это сбой; обычно это предпочтительнее неправильного результата.

C++ Core Guidelines [CG] предлагают правила, позволяющие избежать этого, и советы по статической проверке того, что этого никогда не произойдет. Однако вот несколько подходов, позволяющих избежать проблем с указателями:

- Не сохраняйте указатель на локальный объект после того, как объект выходит за пределы области видимости. В частности, никогда не возвращайте указатель на локальный объект из функции и не храните указатель неопределенного происхождения в долговечных структурах данных. Систематическое использование контейнеров и алгоритмов (глава 12, глава 13) часто избавляет нас от использования методов программирования, из-за которых трудно избежать проблем с указателями.
- Используйте собственные указатели на объекты, размещенные в динамической памяти.

- Указатели на статические объекты (например, глобальные переменные) не могут повиснуть.
- Оставьте арифметику указателей для реализации дескрипторов ресурсов (таких как `vector` и `unordered_map`).
- Помните, что `string_view` и `span` - это разновидности указателей, не являющихся владельцами.

15.2.1 `unique_ptr` и `shared_ptr`

Одной из ключевых задач любой нетривиальной программы является управление ресурсами. Ресурс - это то, что должно быть приобретено и позже (явно или неявно) освобождено. Это может быть память, блокировки, сокет, дескрипторы потоков и дескрипторы файлов. Для длительно работающей программы несвоевременное высвобождение ресурса (“утечка”) может привести к серьезному снижению производительности (§12.7) и, возможно, даже к серьезному сбою. Даже для коротких программ утечка может стать затруднительной, скажем, вызвав нехватку ресурсов, и увеличив время выполнения на порядки.

Компоненты стандартной библиотеки сконструированы таким образом, чтобы не допускать утечки ресурсов. Для этого они полагаются на поддержку базового языка для управления ресурсами с использованием пар конструктор/деструктор, чтобы гарантировать, что ресурс не переживет ответственный за него объект. Примером является использование пары конструктор/деструктор в `Vector` для управления временем жизни его элементов (§5.2.2), и все контейнеры стандартной библиотеки реализованы аналогичным образом. Важно отметить, что этот подход корректно взаимодействует с обработкой ошибок с использованием исключений. Например, этот метод используется для классов блокировки стандартной библиотеки:

```
mutex m; // used to protect access to shared data

void f()
{
    scoped_lock lck {m};          // acquire the mutex m
    // ... manipulate shared data ...
}
```

`thread` не будет запущен до тех пор, пока конструктор `lck` не получит `mutex` (§18.3). Соответствующий деструктор освобождает `mutex`. Итак, в этом примере деструктор `scoped_lock` освобождает `mutex`, когда поток управления покидает `f()` (через `return`, путем “выпадения из конца функции” или посредством выброса исключения).

Здесь реализованно RAII (идиома “Получение ресурса - это инициализация”; §5.2.2). RAII имеет фундаментальное значение для идиоматической обработки ресурсов в C++. Контейнеры (такие как `vector` и `map`, `string` и `iostream`) управляют своими ресурсами (такими как файловые дескрипторы и буферы) аналогичным образом.

Приведенные до сих пор примеры заботятся об объектах, определенных в области видимости, освобождая ресурсы, которые они получают при выходе из области видимости, но как насчет объектов, размещенных в динамической памяти? В `<memory>` стандартная библиотека предоставляет два “умных указателя”, помогающих управлять объектами в динамической памяти:

- `unique_ptr` представляет собой уникальное владение (его деструктор уничтожает его объект)

- `shared_ptr` представляет собой совместное владение (деструктор последнего общего указателя уничтожает объект)

Самое основное применение этих “умных указателей” заключается в предотвращении утечек памяти, вызванных небрежным программированием. Например:

```
void f(int i, int j)                // X* vs. unique_ptr<X>
{
    X* p = new X;                   // allocate a new X
    unique_ptr<X> sp {new X};        // allocate a new X and give its pointer to
    unique_ptr                       // ...

    if (i<99) throw Z{};             // may throw an exception
    if (j<77) return;                // may return "early"
    // ... use p and sp ..
    delete p;                        // destroy *p
}
```

Здесь мы “забыли” удалить `p`, если `i<99` или если `j<77`. С другой стороны, `unique_ptr` гарантирует, что его объект будет должным образом уничтожен, каким бы способом мы ни вышли из `f()` (путем создания исключения, выполнения `return` или “выпадения из конца”). По иронии судьбы, мы могли бы решить проблему, просто *не* используя указатель и *не* используя `new`:

```
void f(int i, int j)                // use a local variable
{
    X x;
    // ...
}
```

К сожалению, злоупотребление оператором `new` (а также указателями и ссылками), по-видимому, становится все более серьезной проблемой.

Однако, когда вам действительно нужна семантика указателей, `unique_ptr` - это облегченный механизм, не требующий затрат памяти или процессорного времени по сравнению с правильным использованием встроенного указателя. Его дальнейшее использование включает передачу объектов, выделенных в динамической памяти, в функции и возврат из них:

```
unique_ptr<X> make_X(int i)
    // make an X and immediately give it to a unique_ptr
{
    //... check i, etc. ...
    return unique_ptr<X>{new X{i}};
}
```

`unique_ptr` - это дескриптор отдельного объекта (или массива) во многом таким же образом, как `vector` является дескриптором последовательности объектов. Оба управляют временем жизни других объектов (используя RAII) и оба полагаются на исключение копирования или семантику перемещения, чтобы сделать `return` простым и эффективным (§6.2.2).

`shared_ptr` похож на `unique_ptr`, за исключением того, что `shared_ptr` копируются, а не перемещаются. `shared_ptr` для объекта разделяют право собственности на объект; этот объект уничтожается, когда уничтожается последний из его `shared_ptr`. Например:

```
void f(shared_ptr<fstream>);
void g(shared_ptr<fstream>);

void user(const string& name, ios_base::openmode mode)
```

```

{
    shared_ptr<fstream> fp {new fstream(name, mode)};
    if (!*fp)                // make sure the file was properly opened
        throw No_file{};

    f(fp);
    g(fp);
    // ...
}

```

Теперь файл, открытый конструктором `fp`, будет закрыт последней функцией, которая (явно или неявно) уничтожит копию `fp`. Обратите внимание, что `f()` или `g()` могут вызвать задачу, содержащую копию `fp`, или каким-либо другим способом сохранить копию, которая переживёт `user()`. Таким образом, `shared_ptr` представляет собой форму сборки мусора, которая реализует управление ресурсами основанное на деструкторе объектов в динамической памяти. Это не является ни бесплатным, ни непомерно дорогим, но из-за этого трудно предсказать срок службы общего объекта. Используйте `shared_ptr` только в том случае, если вам действительно нужно совместное владение.

Создание объекта в динамической памяти и последующая передача указателя на него умному указателю - это немного многословно. Это также допускает ошибки, можно забыть передать указатель на `unique_ptr` или передать указатель на что-то, чего нет в динамической памяти, в `shared_ptr`. Чтобы избежать подобных проблем, стандартная библиотека (в `<memory>`) предоставляет функции для построения объекта и возврата соответствующего интеллектуального указателя, `make_shared()` и `make_unique()`. Например:

```

struct S {
    int i;
    string s;
    double d;
    // ...
};
auto p1 = make_shared<S>(1, "Ankh Morpork", 4.65);    // p1 is a shared_ptr<S>
auto p2 = make_unique<S>(2, "Oz", 7.62);             // p2 is a unique_ptr<S>

```

Теперь `p2` - это `unique_ptr<S>`, указывающий на выделенный в динамической памяти объект типа `S` со значением `{2, "Oz", 7.62}`.

Использование `make_shared()` не просто удобнее, чем отдельное создание объекта с помощью `new` и последующая передача его в `shared_ptr` – это также заметно более эффективно, поскольку не требует отдельного выделения для подсчета использования, что важно при реализации `shared_ptr`.

Используя `unique_ptr` и `shared_ptr`, мы можем полностью реализовать политику “никаких голых `new`” (§5.2.2) для большинства программ. Однако эти “умные указатели” по-прежнему концептуально являются указателями и, следовательно, являются лишь моим вторым выбором для управления ресурсами - после контейнеров и других типов, которые управляют своими ресурсами на более высоком концептуальном уровне. В частности, `shared_ptr` сами по себе не предоставляют никаких правил, по которым их владельцы могут читать и/или записывать общий объект. Состояние гонки за данными (§18.5) и другие формы путаницы не устраняются простым устранением проблем с управлением ресурсами.

Когда мы используем “умные указатели” (такие как `unique_ptr`) вместо дескрипторов ресурсов с операциями, разработанными специально для ресурса (такими как `vector` или `thread`)? Неудивительно, что ответ таков: “когда нам нужна семантика указателя”.

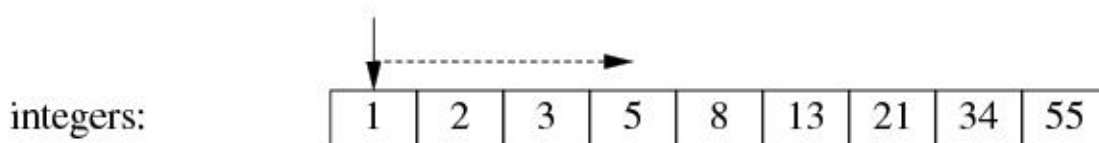
- Когда мы совместно используем объект, нам нужны указатели (или ссылки) для ссылки на общий объект, поэтому `shared_ptr` становится очевидным выбором (если только нет очевидного единственного владельца).
- Когда мы ссылаемся на полиморфный объект в классическом объектно-ориентированном коде (§5.5), нам нужен указатель (или ссылка), потому что мы не знаем точный тип объекта, на который ссылаемся (или даже его размер), поэтому очевидным выбором становится `unique_ptr`.
- Для общего полиморфного объекта обычно требуется `shared_ptr`.

Нам *не* нужно использовать указатель для возврата коллекции объектов из функции; контейнер, который является дескриптором ресурса, делает это просто и эффективно, полагаясь на исключение копирования (§3.4.2) и семантику перемещения (§6.2.2).

15.2.2 `span`

Традиционно ошибки диапазона были основным источником серьезных ошибок в программах на C и C++, приводящих к неправильным результатам, сбоям и проблемам безопасности. Использование контейнеров (глава 12), алгоритмов (глава 13) и цикла `for` для диапазонов значительно уменьшило эту проблему, но можно сделать еще больше. Ключевым источником ошибок диапазона является то, что люди передают указатели (сырые или умные), а затем полагаются на соглашение, чтобы узнать количество элементов, на которые указывают. Лучший совет для кода вне дескрипторов ресурсов - предположить, что не более чем на один объект указывает [CG: F.22], но без поддержки этот совет неосуществим. `string_view` стандартной библиотеки (§10.3) может помочь, но он доступен только для чтения и только для символов. Большинству программистов нужно больше. Например, при записи в буферы и считывании из них в программном обеспечении более низкого уровня, как известно, трудно поддерживать высокую производительность, избегая при этом ошибок диапазона (“перелет буфера”). `span` из `` - это, по сути, пара (указатель, длина), обозначающая последовательность элементов:

`span<int>`: { `begin()` , `size()` }



`span` предоставляет доступ к непрерывной последовательности элементов. Элементы могут храниться многими способами, в том числе в `vector` и встроенных массивах. Как и указатель, `span` не владеет символами, на которые он указывает. В этом он напоминает `string_view` (§10.3) и пару итераторов STL (§13.3).

Рассмотрим общий стиль интерфейса:

```
void fpn(int* p, int n)
{
    for (int i = 0; i < n; ++i)
        p[i] = 0;
}
```


Мы предполагаем, что `p` указывает на `n` целых чисел. К сожалению, это предположение является просто соглашением, поэтому мы не можем использовать его для написания цикла `for` для диапазона, а компилятор не может реализовать дешевую и эффективную проверку диапазона. Кроме того, наше предположение может быть неверным:

```
void use(int x)
{
    int a[100];
    fpn(a,100);           // OK
    fpn(a,1000);          // oops, my finger slipped! (range error in fpn)
    fpn(a+10,100);        // range error in fpn
    fpn(a,x);             // suspect, but looks innocent
}
```

Мы можем добиться большего успеха, используя `span`:

```
void fs(span<int> p)
{
    for (int& x : p)
        x = 0;
}
```

Мы можем использовать `fs` следующим образом:

```
void use(int x)
{
    int a[100];
    fs(a);                // implicitly creates a span<int>{a,100}
    fs(a,1000);            // error: span expected
    fs({a+10,100});        // a range error in fs
    fs({a,x});             // obviously suspect
}
```

То есть обычный случай, когда `span` созданный непосредственно из массива, теперь безопасен (компилятор вычисляет количество элементов) и прост с точки зрения нотации. В других случаях вероятность ошибок снижается, а обнаружение ошибок упрощается, поскольку программисту приходится явно составлять `span`.

Обычный случай, когда `span` передается от функции к функции, проще, чем для интерфейсов (указатель, счетчик), и, очевидно, не требует дополнительной проверки:

```
void f1(span<int> p);

void f2(span<int> p)
{
    // ...
    f1(p);
}
```

Что касается контейнеров, когда для индексации используется `span` (например, `r[i]`), проверка диапазона не выполняется, и доступ вне диапазона является неопределенным поведением. Естественно, на практике можно реализовать это неопределенное поведение как проверку диапазона, но, к сожалению, мало кто это делает. Исходный `gsl::span` из библиотеки поддержки Core Guidelines [CG] выполняет проверку диапазона.

15.3 Контейнеры

Стандарт предоставляет несколько контейнеров, которые не совсем вписываются в структуру STL ([глава 12](#), [глава 13](#)). Примерами являются встроенные массивы, `array` и `string`. Я иногда называю их “почти контейнерами”, но это не совсем справедливо: они

содержат элементы, поэтому они являются контейнерами, но у каждого есть ограничения или дополнительные возможности, которые делают их неудобными в контексте STL. Описание их по отдельности также упрощает описание STL.

Контейнеры	
T[N]	Встроенный массив: непрерывно выделенная последовательность из N элементов фиксированного размера типа T ; неявно преобразуется в T*
array<T,N>	непрерывно выделенная последовательность из N элементов фиксированного размера типа T ; аналогично встроенному массиву, но с решением большинства проблем
bitset<N>	Последовательность фиксированного размера из N бит
vector<bool>	Последовательность бит, компактно хранящаяся в специализированном vector
pair<T,U>	Пара элементов типа T и U
tuple<T...>	Последовательность из произвольного числа элементов произвольных типов
basic_string<C>	Последовательность символов типа C ; обеспечивает операции со строками
valarray<T>	Массив числовых значений типа T ; обеспечивает выполнение числовых операций

Почему в стандарте предусмотрено так много контейнеров? Они удовлетворяют общие, но разные (часто пересекающиеся) потребности. Если бы стандартная библиотека не предоставляла их, многим людям пришлось бы разрабатывать и внедрять свои собственные. Например:

- **pair** и **tuple** неоднородны; все остальные контейнеры однородны (все элементы одного типа).
- элементы **array** и **tuple** располагаются последовательно; **list** и **map** являются связанными структурами.
- **bitset** и **vector<bool>** содержат биты и получают к ним доступ через прокси-объекты; все остальные контейнеры стандартной библиотеки могут содержать различные типы и получать доступ к элементам напрямую.
- **basic_string** требует, чтобы его элементы были некоторой формой символов и обеспечивали манипулирование строками, такое как конкатенация и операции, зависящие от локали.
- **valarray** требует, чтобы его элементы были числами и обеспечивали числовые операции.

Все эти контейнеры можно рассматривать как предоставляющие специализированные услуги, необходимые большим сообществам программистов. Ни один отдельный контейнер не может удовлетворить все эти потребности, потому что некоторые потребности противоречат друг другу, например, “способность к росту” против “гарантированно размещается в фиксированном местоположении” и “элементы не перемещаются при добавлении” против “размещаются в памяти непрерывно”.

15.3.1 array

array, определенный в **<array>**, представляет собой последовательность элементов заданного типа фиксированного размера, где количество элементов задается во время компиляции. Таким образом, **array** может быть размещен вместе со своими элементами в стеке, в объекте или в статическом хранилище. Элементы располагаются в области, в

которой определен `array`. `array` лучше всего понимать как встроенный массив с жестко привязанным к нему размером, без неявных, потенциально неожиданных преобразований в типы указателей и с несколькими предоставляемыми удобными функциями. Использование `array` не требует дополнительных затрат (времени или памяти) по сравнению с использованием встроенного массива. `array` не следует модели “обращения к элементам” контейнеров STL. Вместо этого массив непосредственно содержит свои элементы. Это не что иное, как более безопасная версия встроенного массива.

Это подразумевает, что `array` может и должен быть инициализирован списком инициализаторов:

```
array<int, 3> a1 = {1, 2, 3};
```

Количество элементов в инициализаторе должно быть равно или меньше количества элементов, указанного для `array`.

Количество элементов является обязательным, количество элементов должно быть постоянным выражением, количество элементов должно быть положительным, а тип элемента должен быть явно указан:

```
void f(int n)
{
    array<int> a0 = {1,2,3};           // error: size not specified
    array<string,n> a1 = {"John's", "Queens' "}; // error: size not a constant expression
    array<string,0> a2;                // error: size must be positive
    array<2> a3 = {"John's", "Queens' "}; // error: element type not stated
    // ...
}
```

Если вам нужно, чтобы количество элементов было переменным, используйте `vector`.

При необходимости `array` может быть явно передан функции в стиле C, которая ожидает указатель. Например:

```
void f(int* p, int sz);           // C-style interface

void g()
{
    array<int,10> a;

    f(a,a.size());                // error: no conversion
    f(a.data(),a.size());          // C-style use

    auto p = find(a,777);          // C++/STL-style use (a range is passed)
    // ...
}
```

Зачем нам использовать `array`, когда `vector` намного более гибкий? `array` менее гибок, поэтому он проще. Иногда можно получить значительное преимущество в производительности за счет прямого доступа к элементам, размещенным в стеке, вместо того, чтобы выделять элементы в динамической памяти, получая к ним косвенный доступ через `vector` (дескриптор), а затем освобождая их. С другой стороны, стек - это ограниченный ресурс (особенно в некоторых встроенных системах), и переполнение стека - это неприятно. Кроме того, существуют области применения, такие как безопасное управление в режиме реального времени, где аллокации в динамическую память запрещены. Например, использование `delete` может привести к фрагментации (§12.7) или исчерпанию памяти (§4.3).

Зачем нам использовать `array`, когда мы могли бы использовать встроенный массив? `array` знает свой размер, поэтому его легко использовать с алгоритмами стандартной библиотеки, и его можно скопировать с помощью `=`. Например:

```
array<int,3> a1 = {1, 2, 3 };

auto a2 = a1;      // copy
a2[1] = 5;
a1 = a2;           // assign
```

Однако, главная причина моего предпочтения `array` заключается в том, что он избавляет меня от неожиданных и неприятных преобразований в указатели. Рассмотрим пример, включающий иерархию классов:

```
void h()
{
    Circle a1[10];
    array<Circle,10> a2;
    // ...
    Shape* p1 = a1;      // OK: disaster waiting to happen
    Shape* p2 = a2;      // error: no conversion of array<Circle,10> to Shape*
    (Good!)
    p1[3].draw();        // disaster
}
```

Комментарий "катастрофа" предполагает, что `sizeof(Shape) < sizeof(Circle)`, поэтому обращение по индексу `Circle[]` через `Shape*` дает неправильное смещение. Все стандартные контейнеры обеспечивают правильную итерацию по сравнению со встроенными массивами.

15.3.2 `bitset`

Такие аспекты системы, как состояние входного потока, часто представляются в виде набора флагов, указывающих бинарные условия, такие как "хорошо/плохо", "истина/ложь" и "включено/выключено". C++ эффективно поддерживает понятие небольших наборов флагов с помощью побитовых операций над целыми числами (§1.4). Класс `bitset<N>` обобщает это понятие, предоставляя операции с последовательностью из `N` бит `[0:N)`, где `N` известно во время компиляции. Для наборов битов, которые не помещаются в `long long int` (часто 64 бита), использовать `bitset` гораздо удобнее, чем использовать целые числа напрямую. Для небольших наборов `bitset` обычно оптимизируется. Если вы хотите присвоить битам имена, а не нумеровать их, вы можете использовать `set` (§12.5) или перечисление (§2.4).

`bitset` может быть инициализирован целым числом или строкой:

```
bitset<9> bs1 {"110001111"};
bitset<9> bs2 {0b1'1000'1111};      // binary literal using digit separators (§1.4)
```

Могут быть применены обычные побитовые операторы (§1.4) и операторы сдвига влево и вправо (`<<` и `>>`):

```
bitset<9> bs3 = ~bs1;      // complement: bs3=="001110000"
bitset<9> bs4 = bs1&bs3;   // all zeros
bitset<9> bs5 = bs1<<2;    // shift left: bs5 = "000111100"
```

Операторы сдвига (здесь, `<<`) "сдвигают" нули.

Операции `to_ulong()` и `to_string()` предоставляют конструкторам обратные операции. Например, мы могли бы записать двоичное представление `int`:


```

pair<Entry*,Error_code> complex_search(vector<Entry>& v, const string& s)
{
    Entry* found = nullptr;
    Error_code err = Error_code::found;
    // ... search for s in v ...
    return {found,err};
}

void user(const string& s)
{
    auto r = complex_search(entry_table,s);           // search entry_table
    if (r.second != Error_code::good) {
        // ... handle error ...
    }
    // ... use r.first ....
}

```

Члены **pair** называются **first** и **second**. Это имеет смысл с точки зрения разработчика, но в коде приложения мы можем захотеть использовать наши собственные имена. Для решения этой проблемы можно использовать структурное связывание (§3.4.5):

```

void user(const string& s)
{
    auto [ptr,success] = complex_search(entry_table,s);    // search entry_table
    if (success != Error_code::good)
        // ... handle error ...
    }
    // ... use r.ptr ....
}

```

pair стандартной библиотеки (из **<utility>**) довольно часто используется для вариантов использования “пары значений” в стандартной библиотеке и других местах. Например, алгоритм стандартной библиотеки **equal_range** задает **pair** итераторов, определяющих подпоследовательность, удовлетворяющую предикату:

```

template<typename Forward_iterator, typename T, typename Compare>
pair<Forward_iterator,Forward_iterator>
equal_range(Forward_iterator first, Forward_iterator last, const T& val, Compare
cmp);

```

Учитывая отсортированную последовательность **[first:last)**, функция **equal_range()** вернет **pair**, представляющую подпоследовательность, соответствующую предикату **cmp**. Мы можем использовать это для поиска в отсортированной последовательности из **Record**:

```

auto less = [](const Record& r1, const Record& r2) { return r1.name<r2.name;}; // com-
pare names

void f(const vector<Record>& v)           // assume that v is sorted on its "name"
field
{
    auto [first,last] = equal_range(v.begin(),v.end(),Record{"Reg"},less);

    for (auto p = first; p!=last; ++p)    // print all equal records
        cout << *p;                     // assume that << is defined for Record
}

```

pair предоставляет операторы, такие как **=**, **==** и **<** если это делают ее элементы. Вывод типа позволяет легко создать **pair** без явного указания ее типа. Например:

```

void f(vector<string>& v)
{
    pair p1 {v.begin(),2};           // one way
    auto p2 = make_pair(v.begin(),2); // another way
    // ...
}

```

И `p1`, и `p2` имеют тип `pair<vector<string>::iterator,int>`.

Когда код не обязательно должен быть универсальным, простая структура с именованными элементами часто приводит к созданию более удобного в обслуживании кода.

15.3.4 tuple

Как и массивы, контейнеры стандартной библиотеки однородны, то есть все их элементы относятся к одному типу. Однако иногда мы хотим использовать последовательность элементов разных типов как единый объект; то есть нам нужен гетерогенный контейнер; примером является `pair`, но все такие гетерогенные последовательности содержат только два элемента. Стандартная библиотека предоставляет `tuple` как обобщение `pair` с нулем или более элементов:

```

tuple t0 {}; // empty
tuple<string,int,double> t1 {"Shark",123,3.14}; // the type is explicitly specified
auto t2 = make_tuple(string{"Herring"},10,1.23); // the type is deduced to
                                                    // tuple<string,int,double>
tuple t3 {"Cod"s,20,9.99}; // the type is deduced to
                           // tuple<string,int,double>

```

Элементы `tuple` независимы; между ними не поддерживается инвариант (§4.3). Если нам нужен инвариант, мы должны инкапсулировать кортеж в класс, который его применяет.

Для единственного, конкретного использования простая структура `struct` часто идеальна, но есть много универсальных применений, где гибкость `tuple` избавляет нас от необходимости определять множество структур `struct` за счет отсутствия мнемонических имен для членов. Доступ к членам `tuple` осуществляется через шаблонную функцию `get`. Например:

```

string fish = get<0>(t1); // get the first element: "Shark"
int count = get<1>(t1); // get the second element: 123
double price = get<2>(t1); // get the third element: 3.14

```

Элементы `tuple` нумеруются (начиная с нуля) и аргумент индекс для `get()` должен быть константой. Функция `get` - это шаблонная функция, принимающая индекс в качестве аргумента значения шаблона (§7.2.2).

Доступ к членам `tuple` по их индексу является общим, уродливым и в некоторой степени подверженным ошибкам. К счастью, элемент `tuple` с уникальным типом в этом кортеже может быть "назван" по его типу:

```

auto fish = get<string>(t1); // get the string: "Shark"
auto count = get<int>(t1); // get the int: 123
auto price = get<double>(t1); // get the double: 3.14

```

Можно использовать `get<>` следующим образом:

```

get<string>(t1) = "Tuna"; // write to the string
get<int>(t1) = 7; // write to the int
get<double>(t1) = 312; // write to the double

```


Большинство применений **tuple** скрыто в реализациях конструкций более высокого уровня. Например, мы могли бы получить доступ к элементам **t1**, используя структурное связывание (§3.4.5):

```
auto [fish, count, price] = t1;
cout << fish << ' ' << count << ' ' << price << '\n';    // read
fish = "Sea Bass";                                           // write
```

Как правило, такая привязка и лежащее в ее основе использование **tuple** применяются для вызова функции:

```
auto [fish, count, price] = todays_catch();
cout << fish << ' ' << count << ' ' << price << '\n';
```

Реальная сила **tuple** заключается в том, что вам приходится хранить или передавать неизвестное количество элементов неизвестных типов в качестве объекта.

Явно, перебор элементов **tuple** немного запутан, требуя рекурсии и вычисления тела функции во время компиляции:

```
template <size_t N = 0, typename... Ts>
constexpr void print(tuple<Ts...> tup)
{
    if constexpr (N < sizeof...(Ts)) {        // not yet at the end?
        cout << get<N>(tup) << ' ';          // print the Nth element
        print<N+1>(tup);                      // print the next element
    }
}
```

Здесь, **sizeof...(Ts)** задает количество элементов в **Ts**.

Использовать функцию **print()** очень просто:

```
print(t0);           // no output
print(t2);           // Herring 10 1.23
print(tuple{ "Norah", 17, "Gavin", 14, "Anya", 9, "Courtney", 9, "Ada", 0 }));
```

Как и **pair**, **tuple** предоставляет операторы, такие как **=**, **==** и **<** если это делают его элементы. Существуют также преобразования между **pair** и **tuple** с двумя элементами,

15.4 Альтернативы

Стандарт предлагает три типа альтернативных выражений:

Альтернативы	
union	Встроенный тип, содержащий одну из набора альтернатив (§2.5)
variant<T...>	Один из заданного набора альтернатив (в <variant>)
optional<T>	Значение типа T или отсутствие значения (в <optional>)
any	Значение, одно из неограниченного набора альтернативных типов (в <any>)

Эти типы предлагают пользователю соответствующие функциональные возможности. К сожалению, они не предлагают унифицированного интерфейса.

15.4.1 variant

variant<A, B, C> часто является более безопасной и удобной альтернативой явному использованию **union** (§2.5). Возможно, самым простым примером является возврат либо значения, либо кода ошибки:

```
variant<string, Error_code> compose_message(istream& s)
{
    string mess;
    // ... read from s and compose message ...
    if (no_problems)
        return mess;                                // return a string
    else
        return Error_code{some_problem};            // return an Error_code
}
```

Когда вы присваиваете **variant** значение или инициализируете его, он запоминает тип этого значения. Позже мы сможем узнать, какой тип содержит **variant**, и извлечь значение. Например:

```
auto m = compose_message(cin);

if (holds_alternative<string>(m)) {
    cout << get<string>(m);
}
else {
    auto err = get<Error_code>(m);
    // ... handle error ...
}
```

Этот стиль нравится некоторым людям, которым не нравятся исключения (см. §4.4), но есть и более интересные варианты использования. Например, простому компилятору может потребоваться различать различные типы узлов с различными представлениями:

```
using Node = variant<Expression, Statement, Declaration, Type>;

void check(Node* p)
{
    if (holds_alternative<Expression>(*p)) {
        Expression& e = get<Expression>(*p);
        // ...
    }
    else if (holds_alternative<Statement>(*p)) {
        Statement& s = get<Statement>(*p);
        // ...
    }
    // ... Declaration and Type ...
}
```

Такая схема проверки альтернатив для принятия решения о соответствующих действиях настолько распространена и относительно неэффективна, что заслуживает прямой поддержки:

```
void check(Node* p)
{
    visit(overloaded {
        [](Expression& e) { /* ... */ },
        [](Statement& s) { /* ... */ },
        // ... Declaration and Type ...
    }, *p);
}
```

Это в основном эквивалентно вызову виртуальной функции, но потенциально быстрее. Как и во всех заявлениях о производительности, это “потенциально быстрее” должно быть подтверждено измерениями, когда производительность имеет решающее

значение. Для большинства применений разница в производительности незначительна.

Класс **overloaded** необходим и, как ни странно, не является стандартным. Это “кусочек волшебства”, который создает набор перегрузок из набора аргументов (обычно лямбд):

```
template<class... Ts>
struct overloaded : Ts... {           // variadic template (§8.4)
    using Ts::operator()...;
};

template<class... Ts>
    overloaded(Ts...) -> overloaded<Ts...>;    // deduction guide
```

Затем “посетитель” **visit** применяет **()** к объекту **overload**, который выбирает наиболее подходящую лямбду для вызова в соответствии с правилами вывода перегрузки.

Правила вывода - это механизм для разрешения тонких двусмысленностей, в первую очередь для конструкторов шаблонов классов в базовых библиотеках (§7.2.3).

Если мы попытаемся получить доступ к **variant**, содержащему тип, отличный от ожидаемого, будет брошено исключение **bad_variant_access**.

15.4.2 optional

optional<A> можно рассматривать как особый вид **variant** (например, **variant<A, nothing>**) или как обобщение идеи **A***, либо указывающего на объект, либо являющегося **nullptr**.

optional может быть полезен для функций, которые могут возвращать, а могут и не возвращать объект:

```
optional<string> compose_message(istream& s)
{
    string mess;
    // ... read from s and compose message ...
    if (no_problems)
        return mess;
    return {};           // the empty optional
}
```

Учитывая это, мы можем написать:

```
if (auto m = compose_message(cin))
    cout << *m;           // note the dereference (*)
else {
    // ... handle error ...
}
```

Это нравится некоторым программистам, которым не нравятся исключения (см. §4.4). Обратите внимание на любопытное использование *****. **optional** обрабатывается как указатель на его объект, а не как сам объект.

В **optional** эквивалентом **nullptr** является пустой объект **{}**. Например:

```
int sum(optional<int> a, optional<int> b)
{
    int res = 0;
    if (a) res+=*a;
    if (b) res+=*b;
    return res;
}
```

```
int x = sum(17,19);           // 36
int y = sum(17,{});          // 17
int z = sum({},{});          // 0
```

Если мы попытаемся получить доступ к **optional**, который не содержит значения, результат будет неопределенным; исключение *не* генерируется. Таким образом, **optional** не гарантирует типобезопасность. Не пытайтесь:

```
int sum2(optional<int> a, optional<int> b)
{
    return *a*b;           // asking for trouble
}
```

15.4.3 any

any может содержать произвольный тип и знать, какой тип (если таковой имеется) он содержит. По сути, это неограниченная версия **variant**:

```
any compose_message(istream& s)
{
    string mess;

    // ... read from s and compose message ...

    if (no_problems)
        return mess;           // return a string
    else
        return error_number;    // return an int
}
```

Когда вы присваиваете или инициализируете **any** значением, он запоминает тип этого значения. Позже мы сможем извлечь значение, хранящееся в **any**, указав ожидаемый тип значения. Например:

```
auto m = compose_message(cin);
string& s = any_cast<string>(m);
cout << s;
```

Если мы попытаемся получить доступ к **any**, содержащему тип, отличный от ожидаемого, будет брошено исключение **bad_any_access**.

15.5 Советы

- [1] Библиотека не обязательно должна быть большой или сложной, чтобы быть полезной; §16.1.
- [2] Ресурс - это все, что должно быть приобретено и (явно или неявно) освобождено; §15.2.1.
- [3] Используйте дескрипторы для управления ресурсами (RAII); §15.2.1; [CG: R.1].
- [4] Проблема с **T*** заключается в том, что он может использоваться для представления чего угодно, поэтому мы не можем легко определить назначение “сырого” указателя; §15.2.1.
- [5] Используйте **unique_ptr** для ссылки на объекты полиморфного типа; §15.2.1; [CG: R.20].
- [6] Используйте **shared_ptr** для ссылки (только) на общие объекты; §15.2.1; [CG: R.20].

- [7] Предпочитайте дескрипторы ресурсов со специальной семантикой умным указателям; §15.2.1.
- [8] Не используйте умный указатель там, где подойдет локальная переменная; §15.2.1.
- [9] Предпочитительнее использование `unique_ptr` чем `shared_ptr`; §6.3, §15.2.1.
- [10] Используйте `unique_ptr` или `shared_ptr` в качестве аргументов или возвращаемых значений только для передачи прав собственности; §15.2.1; [CG: F.26] [CG: F.27].
- [11] Используйте `make_unique()` для создания `unique_ptr`; §15.2.1; [CG: R.22].
- [12] Используйте `make_shared()` для создания `shared_ptr`; §15.2.1; [CG: R.23].
- [13] Предпочитайте умные указатели сборке мусора; §6.3, §15.2.1.
- [14] Предпочитительнее `span`, чем интерфейсы типа указатель плюс счетчик; §15.2.2; [CG: F.24].
- [15] `span` поддерживает `for` для диапазонов; §15.2.2.
- [16] Используйте `array` там, где вам нужна последовательность с размером `constexpr`; §15.3.1.
- [17] Предпочитайте `array` встроенным массивам; §15.3.1; [CG: SL.con.2].
- [18] Используйте `bitset`, если вам нужно `N` битов, а `N` - это не обязательно количество бит во встроенном целочисленном типе; §15.3.2.
- [19] Не злоупотребляйте `pair` и `tuple`; именованные `struct` часто приводят к более удобному коду; §15.3.3.
- [20] При использовании `pair`, используйте выведение аргумента шаблона или `make_pair()`, чтобы избежать избыточной спецификации типа; §15.3.3.
- [21] При использовании `tuple`, используйте выведение аргумента шаблона или `make_tuple()` чтобы избежать избыточной спецификации типа; §15.3.3; [CG: T.44].
- [22] Предпочитайте `variant` явному использованию `union`; §15.4.1; [CG: C.181].
- [23] При выборе из набора альтернатив с использованием `variant`, рассмотрите возможность использования `visit()` и `overloaded()`; §15.4.1.
- [24] Если для `variant`, `optional` или `any`, возможно более одной альтернативы, проверьте тип перед доступом; §15.4.

Утилиты

*Время потраченное с удовольствием,
не потрачено впустую.*
– Bertrand Russell

- [Введение](#)
- [Время](#)

[Часы](#); [Календари](#); [Временные зоны](#)

- [Адаптация функций](#)

[Лямбды как адапторы](#); `mem_fn()`; `function`

- [Функция типа](#)

[Предикаты типа](#); [Условные свойства](#); [Генераторы типа](#); [Ассоциированные типы](#)

- `source_location`
- `move()` и `forward()`
- [Битовые манипуляции](#)
- [Выход из программы](#)
- [Советы](#)

16.1 Введение

Обозначение библиотечного компонента как "утилиты" не очень информативно. Очевидно, что каждый библиотечный компонент был полезен кому-то, где-то, в какой-то момент времени. Представленные здесь компоненты выбраны потому, что они служат критически важным целям для многих, но их описание не подходит для других мест. Часто они выступают в качестве строительных блоков для более мощных библиотечных средств, включая другие компоненты стандартной библиотеки.

16.2 Время

В `<chrono>` стандартная библиотека предоставляет средства для работы со временем:

- Часы, `time_point`, и `duration` для измерения того, сколько времени занимает какое-либо действие, и в качестве основы для всего, что связано со временем.
- `day`, `month`, `year`, и `weekdays` для отображения `time_point` в нашей повседневной жизни.

- `time_zone` и `zoned_time` для устранения различий в отсчёте времени по всему миру. По сути, каждая крупная система имеет дело с некоторыми из этих объектов.

16.2.1 Часы

Вот простой способ замерить длительность действий:

```
using namespace std::chrono;           // in sub-namespace std::chrono; see §3.3

auto t0 = system_clock::now();
do_work();
auto t1 = system_clock::now();

cout << t1-t0 << "\n";                // default unit: 20223[1/000000000]s
cout << duration_cast<milliseconds>(t1-t0).count() << "ms\n"; // specify unit: 2ms
cout << duration_cast<nanoseconds>(t1-t0).count() << "ns\n";  // specify unit:
2022300ns
```

Часы возвращает `time_point` (определенный момент времени). Вычитание двух `time_point` дает `duration` (длительность, период времени). По умолчанию `<<` для `duration` добавляет указание группы и отображается как суффикс. Различные часы дают свои результаты в различных единицах измерения времени, “тиках часов”, (я использовал меры в сотни наносекунд), поэтому это часто хорошая идея, преобразовать `duration` в соответствующую единицу измерения, что `duration_cast` и делает.

Часы полезны для быстрых измерений. Не делайте заявления об “эффективности” кода, не проводя измерения быстродействия. Предположения о производительности наиболее ненадежны. Быстрые, простые замеры лучше, чем их отсутствие, но производительность современных компьютеров - это сложная тема, поэтому мы должны быть осторожны, чтобы не придавать слишком большое значение нескольким простым измерениям. Всегда проводите замеры многократно, чтобы снизить шансы на получение ошеломляющих редких событий или эффектов кэша.

Пространства имен `std::chrono_literals` определяет суффиксы единиц измерения времени (§6.6). Например:

```
this_thread::sleep_for(10ms+33us);    // wait for 10 milliseconds and 33 microseconds
```

Обычные символьные имена значительно повышают удобочитаемость и делают код более удобным в обслуживании.

16.2.2 Календари

Когда мы имеем дело с повседневными событиями, мы редко используем миллисекунды; мы используем годы, месяцы, дни, часы, секунды и дни недели. Стандартная библиотека поддерживает это. Например:

```
auto spring_day = April/7/2018;
cout << weekday(spring_day) << '\n';           // Sat
cout << format("{:%A}\n", weekday(spring_day)); // Saturday
```

`Sat` - это символьное представление субботы по умолчанию на моем компьютере. Мне не понравилась эта аббревиатура, поэтому я использовал формат (§11.6.2), чтобы получить более длинное название. По непонятным причинам `%A` означает “напишите полное название дня недели”. Естественно, `April` - это месяц; точнее, `std::chrono::Month`. Мы могли бы также сказать


```
auto spring_day = 2018y/April/7;
```

Суффикс **y** используется для того, чтобы отличать годы от простых **int**, которые используются для обозначения дней месяца с номерами от 1 до 31.

Можно указать недопустимые даты. Если вы сомневаетесь, проверьте с помощью **ok()**:

```
auto bad_day = January/0/2024;
if (!bad_day.ok())
    cout << bad_day << " is not a valid day\n";
```

Очевидно, что **ok()** наиболее полезен для дат, полученных в результате вычислений.

Даты составляются путем перегрузки оператора **/** (косая черта) по типам **year**, **month** и **int**. Результирующий тип **Year_month_day** может преобразовываться в/из **time_point** для обеспечения точного и эффективного вычисления дат. Например:

```
sys_days t = sys_days{February/25/2022}; // get a time point with the precision of
days
t += days{7}; // one week after February 25, 2022
auto d = year_month_day(t); // convert the time point back to the cal-
endar

cout << d << '\n'; // 2022-03-04
cout << format("{:%B}/{}/{ }\n", d.month(), d.day(), d.year()); // March/04/2022
```

Этот расчет требует смены месяца и знаний о високосных годах. По умолчанию реализация указывала дату в стандартном формате ISO 8601. Чтобы указать месяц как “март”, мы должны выделить отдельные поля даты и перейти к деталям форматирования (§11.6.2). По непонятным причинам **%B** означает “напишите полное название месяца”.

Такие операции часто могут выполняться во время компиляции и поэтому выполняются на удивление быстро:

```
static_assert(weekday(April/7/2018) == Saturday); // true
```

Календари сложны и неуловимы. Это типично и уместно для “систем”, разработанных для “обычных людей” на протяжении веков, а не программистами для упрощения программирования. Календарная система стандартной библиотеки может быть расширена (и была расширена) для работы с юлианским, исламским, тайским и другими календарями.

16.2.3 Временные зоны

Один из самых сложных вопросов, связанных со временем, - это часовые пояса. Они настолько произвольны, что их трудно запомнить, и время от времени они меняются различными способами, которые не стандартизированы по всему миру. Например:

```
auto tp = system_clock::now(); // tp is a time_point
cout << tp << '\n'; // 2021-11-27 21:36:08.2085095

zoned_time ztp { current_zone(), tp }; // 2021-11-27 16:36:08.2085095 EST
cout << ztp << '\n';

const time_zone est {"Europe/Copenhagen"};
cout << zoned_time{ &est, tp } << '\n'; // 2021-11-27 22:36:08.2085095
GMT+1
```

`time_zone` - это время относительно стандарта (называемого GMT или UTC), используемого `system_clock`. Стандартная библиотека синхронизируется с глобальной базой данных (IANA), чтобы получить правильные ответы. Эта синхронизация может быть автоматической в операционной системе или под контролем системного администратора. Названия часовых поясов представляют собой строки в стиле C вида “континент/крупный город”, такие как `"America/New_York"`, `"Asia/Tokyo"`, `"Africa/Nairobi"`. `zoned_time` - это `time_zone` вместе с `time_point`.

Как и календари, часовые пояса (временные зоны) решают ряд проблем, которые мы должны оставить стандартной библиотеке, а не полагаться на наш собственный код, созданный вручную. Подумайте: в какое время суток в последний день февраля 2024 года в Нью-Йорке изменится дата в Нью-Дели? Когда в 2020 году в Денвере, штат Колорадо, США, закончилось летнее время? Когда наступит следующая високосная секунда? Стандартная библиотека “знает”.

16.3 Адаптация функций

При передаче функции в качестве аргумента, тип аргумента должен точно соответствовать ожидаемому, выраженному в объявлении вызываемой функции. Если предполагаемый аргумент только “почти соответствует ожиданиям”, у нас есть альтернативные способы его корректировки:

- Использование лямбда-выражений (§16.3.1).
- Использование `std::mem_fn()` чтобы создать функциональный объект из функции-члена (§16.3.2).
- Определение функции, которая будет принимать `std::function` (§16.3.3).

Есть много других способов, но обычно лучше всего работает один из этих трех способов.

16.3.1 Лямбды как адапторы

Рассмотрим классический пример “нарисуй все фигуры”:

```
void draw_all(vector<Shape*>& v)
{
    for_each(v.begin(),v.end(),[](Shape* p) { p->draw(); });
}
```

Как и все алгоритмы стандартной библиотеки, `for_each()` вызывает свой аргумент, используя традиционный синтаксис вызова функции `f(x)`, но `draw()` для `Shape` использует обычную объектно-ориентированную нотацию `x->f()`. Лямбда-выражение легко служит посредником между этими двумя обозначениями.

16.3.2 `mem_fn()`

Учитывая функцию-член, адаптер функции `mem_fn(mf)` создает функциональный объект, который может быть вызван как функция, не являющаяся членом. Например:

```
void draw_all(vector<Shape*>& v)
{
    for_each(v.begin(),v.end(),mem_fn(&Shape::draw));
}
```

До появления лямбд в C++11, `mem_fn()` и эквиваленты были основным способом перехода от объектно-ориентированного стиля вызова к функциональному.

16.3.3 function

В стандартной библиотеке `function` - это тип, содержащий любой объект, который вы можете вызвать с помощью оператора вызова `()`. То есть объект типа `function` является функциональным объектом (§7.3.2). Например:

```
int f1(double);
function<int(double)> fct1 {f1};           // initialize to f1

int f2(string);
function fct2 {f2};                       // fct2's type is function<int(string)>

function fct3 = [](Shape* p) { p->draw(); }; // fct3's type is function<void(Shape*)>
```

Для `fct2` я позволяю вывести тип `function` из инициализатора: `int(string)`.

Очевидно, что `function` полезны для обратных вызовов (callback), для передачи операций в качестве аргументов, для передачи функциональных объектов и т.д. Однако это может привести к некоторым накладным расходам во время выполнения по сравнению с прямыми вызовами. В частности, для объекта `function`, размер которого не вычисляется во время компиляции, может произойти выделение динамической памяти с серьезными негативными последствиями для приложений, критически важных для производительности. Готовится решение для C++23: `move_only_function`.

Другая проблема заключается в том, что `function`, будучи объектом, не участвует в перегрузке. Если вам нужно перегрузить функциональные объекты (включая лямбда-выражения), рассмотрите вариант `overloaded` (§15.4.1).

16.4 Функция типа

Функция типа - это функция, которая вычисляется во время компиляции с заданным типом в качестве аргумента или возвращающая тип. Стандартная библиотека предоставляет множество функций ввода, помогающих разработчикам библиотек (и программистам в целом) писать код, который использует преимущества языка, стандартной библиотеки и кода в целом.

Для числовых типов `numeric_limits` из `<limits>` содержит разнообразную полезную информацию (§17.7). Например:

```
constexpr float min = numeric_limits<float>::min(); // smallest positive float
```

Аналогично, размеры объектов могут быть определены с помощью встроенного оператора `sizeof` (§1.4). Например:

```
constexpr int szi = sizeof(int); // the number of bytes in an int
```

В `<type_traits>` стандартная библиотека предоставляет множество функций для запроса свойств типов. Например:

```
bool b = is_arithmetic_v<X>; // true if X is one of the (built-in) arithmetic types
using Res = invoke_result_t<decltype(f)>; // Res is int if f is a function that // returns an int
```

`decltype(f)` - это вызов встроенной функции типа `decltype(f)`, возвращающей объявленный тип своего аргумента; здесь `f`.

Некоторые функции типа создают новые типы на основе входных данных. Например:

```
typename<typename T>
using Store = conditional_t(sizeof(T) < max, On_stack<T>, On_heap<T>);
```

Если первый (логический) аргумент `conditional_t` равен `true`, результатом будет первая альтернатива; в противном случае - вторая. Предполагая, что `On_stack` и `On_heap` предлагают одинаковые функции доступа к `T`, они могут распределять свои `T` так, как указывают их имена. Таким образом, пользователи `Store<X>` могут быть настроены в соответствии с размером объектов `X`. Настройка производительности, обеспечиваемая таким выбором распределения, может быть очень значительной. Это простой пример того, как мы можем создавать наши собственные функции типа, либо на основе стандартных, либо с помощью концептов.

Концепты - это функции типа. Когда они используются в выражениях, они являются специфическими предикатами типа. Например:

```
template<typename F, typename... Args>
auto call(F f, Args... a, Allocator alloc)
{
    if constexpr (invocable<F,alloc,Args...>) // needs an allocator?
        return f(f,alloc,a...);
    else
        return f(f,a...);
}
```

Во многих случаях концепты являются лучшими функциями типа, но большая часть стандартной библиотеки была написана до появления концептов и должна поддерживать кодовые базы, написанные до появления концептов.

Условные обозначения сбивают с толку. Стандартная библиотека использует `_v` для функций типа, возвращающих значения, и `_t` для функций типа, возвращающие типы, это пережиток слабо типизированных времен C и доконцептного C++. Ни одна функция типа стандартной библиотеки не возвращает одновременно тип и значение, поэтому эти суффиксы являются избыточными. С концептами, как в стандартной библиотеке, так и в других местах, суффикс не требуется и не используется.

Функции типов являются частью механизмов C++ для вычислений во время компиляции, которые обеспечивают более жесткую проверку типов и более высокую производительность, чем это было бы возможно без них. Использование функций и концептов типов (глава 8, §14.5) часто называют *метапрограммированием* или (когда речь идет о шаблонах) *шаблонным метапрограммированием*.

16.4.1 Предикаты типа

В `<type_traits>` стандартная библиотека предлагает десятки простых функций типа, называемых *предикатами типа*, которые отвечают на фундаментальные вопросы о типах. Вот небольшая подборка:

Некоторые предикаты типа	
T, A, и U - типы; все предикаты возвращают bool	
<code>is_void_v<T></code>	Является ли T типом void?
<code>is_integral_v<T></code>	Является ли T целочисленным типом?
<code>is_floating_point_v<T></code>	Является ли T типом число с плавающей точкой?
<code>is_class_v<T></code>	Является ли T классом (и не union)?

<code>is_function_v<T></code>	Является ли T функцией (и не функциональным объектом или не указатель на функцию)?
<code>is_arithmetic_v<T></code>	Является ли T численным типом с плавающей точкой или целым?
<code>is_scalar_v<T></code>	Является ли T арифметическим типом, перечислением, указателем или указателем на элемент данных?
<code>is_constructible_v<T, A...></code>	Может ли T быть создан из списка аргументов A... ?
<code>is_default_constructible_v<T></code>	Может ли T быть создан без обязательных аргументов (конструктором по умолчанию)?
<code>is_copy_constructible_v<T></code>	Может ли T быть создан от другого T ?
<code>is_move_constructible_v<T></code>	Может ли T быть скопирован или перемещён в другой T ?
<code>is_assignable_v<T,U></code>	Может ли U быть присвоенным в T ?
<code>is_trivially_copyable_v<T,U></code>	Может ли U быть присвоенным в T без пользовательских операций копирования?
<code>is_same_v<T,U></code>	Является ли T того же типа как U ?
<code>is_base_of_v<T,U></code>	Является ли U производным от T или оба T и U одного типа?
<code>is_convertible_v<T,U></code>	Может ли T быть неявно преобразованным в U ?
<code>is_iterator_v<T></code>	Является ли T типом итератора?
<code>is_invocable_v<T, A...></code>	Может ли T быть вызван со списком аргументов A... ?
<code>has_virtual_destructor_v<T></code>	Имеет ли T виртуальный деструктор?

Одно из традиционных применений этих предикатов заключается в ограничении аргументов шаблона. Например:

```
template<typename Scalar>
class complex {
    Scalar re, im;
public:
    static_assert(is_arithmetic_v<Scalar>, "Sorry, I support only complex of arithmetic types");
    // ...
};
```

Однако это – как и другие традиционные способы использования – проще и элегантнее сделать с помощью концептов:

```
template<Arithmetic Scalar>
class complex {
    Scalar re, im;
public:
    // ...
};
```

Во многих случаях предикаты типа, такие как `is_arithmetic`, скрываются в определении концептов для большей простоты использования. Например:

```
template<typename T>
concept Arithmetic = is_arithmetic_v<T>;
```

Как ни странно, концепта `std::arithmetic` не существует.

Часто мы можем определить концепты, которые являются более общими, чем предикаты типа стандартной библиотеки. Многие предикаты типов стандартной библиотеки применяются только к встроенным типам. Мы можем определить концепт в терминах требуемых операций, как это предлагается в определении `Number` (§8.2.4):

```
template<typename T, typename U = T>
concept Arithmetic = Number<T,U> && Number<U,T>;
```

Чаще всего использование предикатов типа стандартной библиотеки встречается глубоко в реализации фундаментальных сервисов, часто для выделения вариантов оптимизации. Например, часть реализации `std::copy(liter, liter1, liter2)` могла бы оптимизировать важный случай непрерывных последовательностей простых типов, таких как целые числа:

```
template<class T>
void cpy1(T* first, T* last, T* target)
{
    if constexpr (is_trivially_copyable_v<T>)
        memcpy(first, target, (last - first) * sizeof(T));
    else
        while (first != last) *target++ = *first++;
}
```

Эта простая оптимизация превосходит свой неоптимизированный вариант примерно на 50% в некоторых реализациях. Не позволяйте себе подобных ухищрений, пока вы не убедитесь, что стандарт уже не работает лучше. Оптимизированный вручную код, как правило, менее удобен в обслуживании, чем более простые альтернативы.

16.4.2 Условные свойства

Рассмотрим определение “умного указателя”:

```
template<typename T>
class Smart_pointer {
    // ...
    T& operator*() const;
    T* operator->() const;    // -> should work if and only if T is a class
};
```

Оператор `->` должен быть определен тогда и только тогда, когда `T` является типом класса. Например, `Smart_pointer<vector<T>>` должен иметь `->`, но `Smart_pointer<int>` не должен.

Мы не можем использовать `if` времени компиляции, потому что мы не находимся внутри функции. Вместо этого мы пишем

```
template<typename T>
class Smart_pointer {
    // ...
    T& operator*() const;
    T* operator->() const requires is_class_v<T>;    // -> is defined if and only if T
is a class
};
```

Предикат типа непосредственно выражает ограничение на `operator->()`. Мы также можем использовать концепты для этого. Не существует концепта стандартной библиотеки, требующего, чтобы тип был типом класса (то есть `class`, `struct` или `union`), но мы могли бы определить один из них:

```
template<typename T>
concept Class = is_class_v<T> || is_union_v<T>;    // unions are classes
```

```
template<typename T>
class Smart_pointer {
    // ...
    T& operator*() const;
    T* operator->() const requires Class<T>; // -> is defined if and only if T is a
class or a union
};
```

Часто концепт является более общим или просто более подходящим, чем прямое использование предиката типа стандартной библиотеки.

16.4.3 Генераторы типов

Многие функции типа возвращают типы, часто новые типы, которые они вычисляют. Я называю такие функции *генераторами типов*, чтобы отличать их от предикатов типов. Стандарт предлагает некоторые, например, такие:

Некоторые генераторы типов	
<code>R = remove_const_t<T></code>	<code>R</code> это <code>T</code> с удалением константности (если таковая имеется)
<code>R = add_const_t<T></code>	<code>R</code> это <code>const T</code>
<code>R = remove_reference_t<T></code>	если <code>T</code> это ссылка <code>U&</code> , <code>R</code> это <code>U</code> иначе <code>T</code>
<code>R = add_lvalue_reference_t<T></code>	если <code>T</code> это lvalue ссылка, <code>R</code> это <code>T</code> иначе <code>T&</code>
<code>R = add_rvalue_reference_t<T></code>	если <code>T</code> это rvalue ссылка, <code>R</code> это <code>T</code> иначе <code>T&&</code>
<code>R = enable_if_t<b,T =void></code>	если <code>b</code> истинно, <code>R</code> это <code>T</code> иначе <code>R</code> не определено
<code>R = conditional_t<b,T,U></code>	<code>R</code> это <code>T</code> если <code>b</code> истинно; иначе <code>U</code>
<code>R = common_type_t<T...></code>	если существует тип, к которому все <code>T</code> могут быть неявно преобразованы, <code>R</code> является этим типом; в противном случае <code>R</code> не определен
<code>R = underlying_type_t<T></code>	если <code>T</code> является перечислением, <code>R</code> является его базовым типом; в противном случае ошибка
<code>R = invoke_result_t<T,A...></code>	если <code>T</code> может быть вызван с аргументами <code>A...</code> , <code>R</code> это тип возвращаемого значения; иначе ошибка

Функции такого типа обычно используются при реализации утилит, а не непосредственно в коде приложения. Из них `enable_if`, вероятно, является наиболее распространенным в коде до создания концептов. Например, условно включенный `->` для умного указателя традиционно реализуется примерно так:

```
template<typename T>
class Smart_pointer {
    // ...
    T& operator*();
    enable_if<is_class_v<T>,T&> operator->(); // -> is defined if and only if T is a
class
};
```

Я не нахожу это особенно легким для чтения, а более сложные варианты использования гораздо хуже. Определение `enable_if` основывается на тонкой языковой функции, называемой SFINAE (“Сбой замены не является ошибкой”). Посмотрите это (только), если вам нужно.

16.4.4 Связанные типы

Все стандартные контейнеры (§12.8) и все контейнеры, разработанные в соответствии с их шаблоном, имеют некоторые *связанные типы*, такие как их типы значений и типы итераторов. В `<iterator>` и `<ranges>` стандартная библиотека предоставляет имена для этих:

Некоторые генераторы типов	
<code>range_value_t<R></code>	Тип диапазона элементов <code>R</code>
<code>iter_value_t<T></code>	Тип элементов на которые указывает итератор <code>T</code>
<code>iterator_t<R></code>	Тип итератора диапазона <code>R</code>

16.5 `source_location`

При записи сообщения трассировки или сообщения об ошибке мы часто хотим вывести данные о местоположении ошибки в исходнике частью этого сообщения. Стандартная библиотека предоставляет `source_location` для этого:

```
const source_location loc = source_location::current();
```

Эта функция `current()` возвращает значение `source_location`, описывающее место в исходном коде, где оно появляется. Класс `source_location` имеет элементы `file()` и `function_name()`, возвращающие строки в стиле C, а элементы `line()` и `column()`, возвращающие целые числа без знака.

Оберните это в функцию, и мы получим хороший первый фрагмент сообщения журнала:

```
void log(const string& mess = "", const source_location loc = source_location::current())
{
    cout << loc.file_name()
          << '(' << loc.line() << ':' << loc.column() << ") "
          << loc.function_name() ": "
          << mess;
}
```

Вызов `current()` является аргументом по умолчанию, так что мы получаем местоположение вызывающего `log()`, а не местоположение `log()`:

```
void foo()
{
    log("Hello");           // myfile.cpp (17,4) foo: Hello
    // ...
}

int bar(const string& label)
{
    log(label);             // myfile.cpp (23,4) bar: <<the value of label>>
    // ...
}
```

Код, написанный до C++20 или нуждающийся в компиляции на более старых компиляторах, использует для этого макросы `__FILE__` и `__LINE__`.

16.6 `move()` and `forward()`

Выбор между перемещением и копированием в основном выполняется неявно (§3.4). Компилятор предпочитает перемещение, когда объект вот-вот будет уничтожен

(как при `return`), потому что предполагается, что это более простая и эффективная операция. Однако иногда мы должны быть откровенны. Например, `unique_ptr` является единственным владельцем объекта. Следовательно, он не может быть скопирован, поэтому, если вам нужен `unique_ptr` в другом месте, вы должны переместить его. Например:

```
void f1()
{
    auto p = make_unique<int>(2);
    auto q = p;           // error: we can't copy a unique_ptr
    auto q = move(p);     // p now holds nullptr
    // ...
}
```

Сбивает с толку то, что `std::move()` ничего не перемещает. Вместо этого он приводит свой аргумент к `rvalue` ссылке, тем самым сообщая, что его аргумент больше не будет использоваться и, следовательно, может быть перемещен (§6.2.2). Это должно было называться как-то вроде `rvalue_cast`. Он существует для обслуживания нескольких важных случаев. Рассмотрим простой обмен:

```
template <typename T>
void swap(T& a, T& b)
{
    T tmp {move(a)};      // the T constructor sees an rvalue and moves
    a = move(b);          // the T assignment sees an rvalue and moves
    b = move(tmp);        // the T assignment sees an rvalue and moves
}
```

Мы не хотим повторно копировать потенциально большие объекты, поэтому мы запрашиваем перемещения с помощью `std::move()`.

Что касается других приведений, то существуют заманчивые, но опасные варианты использования `std::move()`. Рассмотрим:

```
string s1 = "Hello";
string s2 = "World";
vector<string> v;
v.push_back(s1);          // use a "const string&" argument; push_back() will copy
v.push_back(move(s2));    // use a move constructor
v.emplace_back(s1);       // an alternative; place a copy of s1 in a new end position
                          // of v (§12.8)
```

Здесь `s1` копируется (с помощью `push_back()`), тогда как `s2` перемещается. Это иногда (только иногда) делает `push_back()` из `s2` дешевле. Проблема в том, что перемещенный объект остается позади. Если мы снова используем `s2`, у нас возникнет проблема:

```
cout << s1[2];           // write 'l'
cout << s2[2];           // crash?
```

Я считаю, что такое использование `std::move()` слишком подвержено ошибкам. Не используйте его, если только вы не сможете продемонстрировать значительное и необходимое улучшение производительности. Последующее техническое обслуживание может случайно привести к непредвиденному использованию перемещенного объекта.

Компилятор знает, что возвращаемое значение больше не используется в функции, поэтому использование явного `std::move()`, например, `return std::move(x)`, является избыточным и может даже препятствовать оптимизации.

Состояние перемещаемого объекта, как правило, не определено, но все типы стандартных библиотек оставляют перемещаемый объект в состоянии, в котором он может быть уничтожен и присвоен. Было бы неразумно не последовать этому примеру. Для контейнера (например, `vector` или `string`) состояние перемещения из будет “пустым”. Для

многих типов хорошим значением по умолчанию является пустое состояние: значимое и дешевое в установке.

Передача аргументов - важный вариант использования, требующий перемещений (§8.4.2). Иногда мы хотим передать набор аргументов в другую функцию, ничего не меняя (для достижения “идеальной пересылки”):

```
template<typename T, typename... Args>
unique_ptr<T> make_unique(Args&&... args)
{
    return unique_ptr<T>{new T{std::forward<Args>(args)...}};    // forward each argu-
ment
}
```

Функция `forward()` стандартной библиотеки отличается от более простой `std::move()` правильной обработкой тонкостей, связанных с lvalue и rvalue (§6.2.2). Используйте `std::forward()` исключительно для передачи и не `forward()` что-либо дважды; как только вы передали объект, он больше не принадлежит вам и нельзя его использовать.

16.7 Битовые манипуляции

В `<bit>` мы находим функции для низкоуровневой обработки битов. Манипулирование битами - это специализированный, но часто необходимый вид деятельности. Когда мы приближаемся к аппаратному обеспечению, нам часто приходится смотреть на биты, изменять битовые структуры в байте или слове и превращать необработанную память в типизированные объекты. Например, `bit_cast` позволяет нам преобразовать значение одного типа в другой тип того же размера:

```
double val = 7.2;
auto x = bit_cast<uint64_t>(val);    // get the bit representation of a 64-bit float-
ing
                                     // point number
auto y = bit_cast<uint64_t>(&val);    // get the bit representation of a 64-bit
pointer

struct Word { std::byte b[8]; };
std::byte buffer[1024];
// ...
auto p = bit_cast<Word*>(&buffer[i]);    // p points to 8 bytes
auto i = bit_cast<int64_t>(*p);    // convert those 8 bytes to an integer
```

Тип стандартной библиотеки `std::byte` (требуется `std::`) существует для представления байтов, а не байтов представляющих символы или целые числа. В частности, `std::byte` предоставляет только побитовые логические операции, а не арифметические. Обычно лучшим типом для выполнения битовых операций является беззнаковое целое или `std::byte`. Под лучшим я подразумеваю самый быстрый и наименее подверженный сюрпризам. Например:

```
void use(unsigned int ui)
{
    int x0 = bit_width(ui)    // the smallest number of bits needed to represent
ui
    unsigned int ui2 = rotl(ui,8)    // rotate left 8 bits (note: doesn't change ui)
    int x1 = popcount(ui);    // the number of 1s in ui
    // ...
}
```

Смотрите также `bitset` (§15.3.2).

16.8 Выход из программы

Иногда фрагмент кода сталкивается с проблемой, с которой он не может справиться:

- Если проблема такого рода возникает часто и можно ожидать, что непосредственный вызывающий справится с ней, верните какой-нибудь код возврата (§4.4).
- Если проблема такого рода возникает нечасто или нельзя ожидать, что непосредственный вызывающий абонент справится с ней, создайте исключение (§4.4).
- Если проблема настолько серьезна, что нельзя ожидать, что ни одна обычная часть программы не справится с ней, выйдите из программы.

Стандартная библиотека предоставляет средства для работы с этим последним случаем (“выход из программы”):

- **exit(x)**: вызов функций, зарегистрированных с помощью **atexit()** затем выход из программы с возвращаемым значением **x**. Если вам нужно, посмотрите **atexit()**, это, по сути, примитивный механизм деструктора, совместно используемый с языком C.
- **abort()**: немедленно и безоговорочно завершить работу программы с возвращаемым значением, указывающим на неудачное завершение. Некоторые операционные системы предлагают средства, которые изменяют это простое поведение.
- **quick_exit(x)**: вызывает функции, зарегистрированные с помощью **at_quick_exit()**; затем завершает работу программы с возвращаемым значением **x**.
- **terminate()**: вызывает **terminate_handler**. По умолчанию **terminate_handler** это **abort()**.

Эти функции предназначены для действительно серьезных ошибок. Они не вызывают деструкторы; то есть они не выполняют обычную и надлежащую очистку. Различные обработчики используются для выполнения действий перед выходом. Такие действия должны быть очень простыми, потому что одной из причин вызова этих функций выхода является то, что состояние программы повреждено. Одно разумное и достаточно популярное действие - “перезапустить систему в четко определенном состоянии, не полагаясь ни на какое состояние текущей программы”. Другое, немного более сложное, но часто небезосновательное действие - “зарегистрировать сообщение об ошибке и выйти”. Причина, по которой запись сообщения журнала может быть проблемой, заключается в том, что система ввода-вывода могла быть повреждена из-за того, что вызвало вызов функции выхода.

Обработка ошибок - один из самых сложных видов программирования; даже чисто выйти из программы может быть непросто.

Ни одна библиотека общего назначения не должна безоговорочно завершать работу.

16.9 Советы

- [1] Библиотека не обязательно должна быть большой или сложной, чтобы быть полезной; §16.1.
- [2] Замерьте время работы ваших программ, прежде чем делать заявления об эффективности; §16.2.1.
- [3] Используйте **duration_cast** чтобы получить измерения времени в соответствующих единицах; §16.2.1.

- [4] Чтобы представить дату непосредственно в исходном коде, используйте символическую запись (например, `November/28/2021`); §16.2.2.
- [5] Если дата является результатом вычисления, проверьте правильность с помощью `ok()`; §16.2.2.
- [6] При работе со временем в разных местах используйте `zoned_time`; §16.2.3.
- [7] Используйте лямбду для согласования незначительных изменений в соглашениях о вызовах; §16.3.1.
- [8] Используйте `mem_fn()` или лямбда-выражение для создания функциональных объектов, которые могут вызывать функцию-член при вызове с использованием традиционной нотации вызова функции; §16.3.1, §16.3.2.
- [9] Используйте `function` когда вам нужно сохранить что-то, что может быть вызвано; §16.3.3.
- [10] Предпочитайте концепты явному использованию предикатов типа; §16.4.1.
- [11] Вы можете написать код, который явно зависит от свойств типов; §16.4.1, §16.4.2.
- [12] Предпочитайте концепты `type_trait` и `enable_if` если можете; §16.4.3.
- [13] Используйте `source_location` для встраивания местоположений исходного кода в сообщения отладки и ведения журнала; §16.5.
- [14] Избегайте явного использования `std::move()`; §16.6; [CG: ES.56].
- [15] Используйте `std::forward()` исключительно для пересылки; §16.6.
- [16] Никогда не считывайте данные из объекта после выполнения `std::move()` или `std::forward()`; §16.6.
- [17] Используйте `std::byte` для представления данных, которые (пока) не имеют значимого типа; §16.7.
- [18] Используйте `unsigned` целые или `bitset` для битовых манипуляций §16.7.
- [19] Возвращайте код ошибки из функции, если можно ожидать, что непосредственный вызывающий объект справится с проблемой; §16.8.
- [20] Бросайте исключение из функции, если нельзя ожидать, что непосредственный вызывающий справится с проблемой; §16.8.
- [21] Вызывайте `exit()`, `quick_exit()`, или `terminate()` для выхода из программы, если попытка устранения неполадки нецелесообразна; §16.8.
- [22] Ни одна библиотека общего назначения не должна безоговорочно завершать работу; §16.8.

Числовые вычисления

Цель вычислений - понимание, а не цифры.

– R. W. Hamming

... но для студента цифры часто являются лучшим путем к пониманию.

– A. Ralston

- [Введение](#)
- [Математические функции](#)
- [Численные алгоритмы](#)

[Многопоточные численные алгоритмы](#)

- [Комплексные числа](#)
- [Случайные числа](#)
- [Векторная арифметика](#)
- [Числовые ограничения](#)
- [Псевдонимы чисел](#)
- [Математические константы](#)
- [Советы](#)

17.1 Введение

C++ не был разработан в первую очередь с учетом числовых вычислений. Однако числовые вычисления обычно выполняются в контексте другой работы – такой, как научные вычисления, доступ к базам данных, создание сетей, управление приборами, графика, моделирование и финансовый анализ, – поэтому C++ становится привлекательным средством для вычислений, которые являются частью более крупной системы. Кроме того, числовые методы прошли долгий путь от простых циклов над векторами чисел с плавающей запятой. Там, где в рамках вычислений требуются более сложные структуры данных, преимущества C++ становятся актуальными. Конечным результатом является то, что C++ широко используется для научных, инженерных, фи-

нансовых и других вычислений, связанных со сложными числами. Следовательно, появились средства и методы, поддерживающие такие вычисления. В этой главе описываются части стандартной библиотеки, поддерживающие численные методы.

17.2 Математические функции

В `<cmath>` мы находим *стандартные математические функции*, такие как `sqrt()`, `log()` и `sin()` для аргументов типа `float`, `double` и `long double`:

Некоторые стандартные математические функции	
<code>abs(x)</code>	Абсолютное значение
<code>ceil(x)</code>	Округление до целого в большую сторону $\geq x$
<code>floor(x)</code>	Округление до целого в меньшую сторону $\leq x$
<code>sqrt(x)</code>	Квадратный корень; x не отрицательный
<code>cos(x)</code>	Косинус
<code>sin(x)</code>	Синус
<code>tan(x)</code>	Тангенс
<code>acos(x)</code>	Аркосинус; результат не отрицательный
<code>asin(x)</code>	Арсинус; возвращается результат, ближайший к 0
<code>atan(x)</code>	Арктангенс
<code>sinh(x)</code>	Гиперболический синус
<code>cosh(x)</code>	Гиперболический косинус
<code>tanh(x)</code>	Гиперболический тангенс
<code>exp(x)</code>	Экспонента по основанию e
<code>exp2(x)</code>	Экспонента по основанию 2
<code>log(x)</code>	Натуральный логарифм по основанию e ; x должен быть положительным
<code>log2(x)</code>	Натуральный логарифм по основанию 2; x должен быть положительным
<code>log10(x)</code>	Логарифм по основанию 10; x должен быть положительным

Версии этих функций для `complex` (§17.4) приведены в `<complex>`. Для каждой функции возвращаемый тип совпадает с типом аргумента.

Об ошибках сообщается путем установки `errno` из `<cerrno>` в `EDOM` для ошибки домена и в `ERANGE` для ошибки диапазона. Например:

```
errno = 0;           // clear old error state
double d = sqrt(-1)
if (errno==EDOM)
    cerr << "sqrt() not defined for negative argument\n";

errno = 0;           // clear old error state
double dd = pow(numeric_limits<double>::max(),2);
if (errno == ERANGE)
    cerr << "result of pow() too large to represent as a double\n";
```

Дополнительные математические функции можно найти в `<cmath>` и `<cstdlib>`. Так называемые *специальные математические функции*, такие как `beta()`, `rieman_zeta()` и `sph_bessel()`, также находятся в `<cmath>`.

17.3 Численные алгоритмы

В `<numeric>` мы находим небольшой набор обобщенных численных алгоритмов, таких как `accumulate()`.

Численные алгоритмы	
<code>x=accumulate(b,e,i)</code>	<code>x</code> - это сумма <code>i</code> и элементов <code>[b:e]</code>
<code>x=accumulate(b,e,i,f)</code>	<code>accumulate</code> используя функцию <code>f</code> вместо <code>+</code>
<code>x=inner_product(b,e,b2,i)</code>	<code>x</code> является внутренним продуктом <code>[b:e]</code> и <code>[b2:b2+(e-b))</code> , то есть сумма <code>i</code> и <code>(*p1)*(*p2)</code> для каждого <code>p1</code> в <code>[b:e]</code> и соответствующего <code>p2</code> в <code>[b2:b2+(e-b))</code>
<code>x=inner_product(b,e,b2,i,f,f2)</code>	<code>inner_product</code> используя функции <code>f</code> и <code>f2</code> вместо <code>+</code> и <code>*</code>
<code>p=partial_sum(b,e,out)</code>	Элемент <code>i</code> из <code>[out:p]</code> является суммой элементов <code>[b:b+i]</code>
<code>p=partial_sum(b,e,out,f)</code>	<code>partial_sum</code> используя функцию <code>f</code> вместо <code>+</code>
<code>p=adjacent_difference(b,e,out)</code>	Элемент <code>i</code> из <code>[out:p]</code> это <code>*(b+i)-*(b+i-1)</code> для <code>i>0</code> ; если <code>e-b>0</code> , тогда <code>*out</code> это <code>*b</code>
<code>p=adjacent_difference(b,e,out,f)</code>	<code>adjacent_difference</code> используя функцию <code>f</code> вместо <code>-</code>
<code>iota(b,e,v)</code>	Для каждого элемента в <code>[b:e]</code> присваивает <code>v</code> и увеличивает <code>++v</code> ; получаем такую последовательность <code>v, v+1, v+2, ...</code>
<code>x=gcd(n,m)</code>	<code>x</code> является наибольшим общим знаменателем целых чисел <code>n</code> и <code>m</code>
<code>x=lcm(n,m)</code>	<code>x</code> является наименьшим общим кратным целых чисел <code>n</code> и <code>m</code>
<code>x=midpoint(n,m)</code>	<code>x</code> является средней точкой между <code>n</code> и <code>m</code>

Эти алгоритмы обобщают распространенные операции, такие как вычисление суммы, позволяя им применяться ко всем видам последовательностей. Они также делают операцию, применяемую к элементам этих последовательностей, параметром. Для каждого алгоритма общая версия дополняется версией, применяющей наиболее распространенный оператор для этого алгоритма. Например:

```
list<double> lst {1, 2, 3, 4, 5, 9999.99999};
auto s = accumulate(lst.begin(),lst.end(),0.0);           // calculate the sum:
10014.9999
```

Эти алгоритмы работают для каждой последовательности из стандартной библиотеки и могут содержать операции, предоставляемые в качестве аргументов (§17.3).

17.3.1 Многопоточные численные алгоритмы

В `<numeric>` численные алгоритмы (§17.3) имеют параллельные версии, которые незначительно отличаются от последовательных. В частности, параллельные версии допускают операции с элементами в неопределенном порядке. Параллельные численные алгоритмы могут принимать аргумент политики выполнения (§13.6): `seq`, `unseq`, `par` и `par_unseq`.

Параллельные численные алгоритмы	
<code>x=reduce(b,e,v)</code>	<code>x=accumulate(b,e,v)</code> , кроме вышедших из строя
<code>x=reduce(b,e)</code>	<code>x=reduce(b,e,V{})</code> , где <code>V</code> это тип хранилища для <code>b</code>
<code>x=reduce(pol,b,e,v)</code>	<code>x=reduce(b,e,v)</code> с политикой выполнения <code>pol</code>

<code>x=reduce(pol,b,e)</code>	<code>x=reduce(pol,b,e,V{})</code> , где <code>V</code> это тип хранилища для <code>b</code>
<code>p=exclusive_scan(pol,b,e,out)</code>	<code>p=partial_sum(b,e,out)</code> согласно <code>pol</code> , исключает <code>i</code> -й входной элемент из <code>i</code> -й суммы
<code>p=inclusive_scan(pol,b,e,out)</code>	<code>p=partial_sum(b,e,out)</code> согласно <code>pol</code> включает <code>i</code> -й входной элемент в <code>i</code> -ю сумму
<code>p=transform_reduce(pol,b,e,f,v)</code>	<code>f(x)</code> для каждого <code>x</code> в <code>[b:e)</code> , применяет <code>reduce</code>
<code>p=transform_exclusive_scan(pol,b,e,out,f,v)</code>	<code>f(x)</code> для каждого <code>x</code> в <code>[b:e)</code> , применяет <code>exclusive_scan</code>
<code>p=transform_inclusive_scan(pol,b,e,out,f,v)</code>	<code>f(x)</code> для каждого <code>x</code> в <code>[b:e)</code> , применяет <code>inclusive_scan</code>

Для простоты я опустил версии этих алгоритмов, которые принимают операции в качестве аргументов, а не просто используют `+` и `=`. За исключением `reduce()`, я также опустил версии с политикой по умолчанию (последовательной) и значением по умолчанию.

Точно так же, как для параллельных алгоритмов в `<algorithm>` (§13.6), мы можем указать политику выполнения:

```
vector<double> v {1, 2, 3, 4, 5, 9999.99999};
auto s = reduce(v.begin(),v.end());           // calculate the sum using a double as
the                                           // accumulator

vector<double> large;
// ... fill large with lots of values ...
auto s2 = reduce(par_unseq,large.begin(),large.end()); // calculate the sum using
// available parallelism
```

Политики выполнения, `par`, `sec`, `unseq` и `par_unseq` скрыты в пространстве имен `std::execution` в `<execution>`.

Измерьте быстродействие, чтобы убедиться в целесообразности использования параллельного или векторизованного алгоритма.

17.4 Комплексные числа

Стандартная библиотека поддерживает семейство типов комплексных чисел в соответствии с классом `complex`, описанным в §5.2.1. Для поддержки комплексных чисел, где скалярами являются числа с плавающей запятой одинарной точности (`float`), числа с плавающей запятой двойной точности (`double`) и т.д., класс `complex` стандартной библиотеки является шаблонным:

```
template<typename Scalar>
class complex {
public:
    complex(const Scalar& re = {}, const Scalar& im = {}); // default function
                                                         // arguments; see §3.4.1
    // ...
};
```

Для комплексных чисел поддерживаются обычные арифметические операции и наиболее распространенные математические функции. Например:

```
void f(complex<float> f1, complex<double> db)
{
    complex<long double> ld {f1+sqrt(db)};
    db += f1*3;
```

```

    f1 = pow(1/f1,2);
    // ...
}

```

Функции `sqrt()` и `pow()` (возведение в степень) относятся к числу обычных математических функций, определенных в `<complex>` (§17.2).

17.5 Случайные числа

Случайные числа полезны во многих контекстах, таких как тестирование, игры, моделирование и безопасность. Разнообразие областей применения отражается в широком выборе генераторов случайных чисел, предоставляемых стандартной библиотекой в `<random>`. Генератор случайных чисел состоит из двух частей:

[1] *Генератор*, который генерирует последовательность случайных или псевдослучайных значений

[2] *Распределение*, которое преобразует эти значения в математическое распределение в диапазоне

Примерами распределений являются `uniform_int_distribution` (где все полученные целые числа имеют равную вероятность), `normal_distribution` (нормальное распределение, “колоколообразная кривая”) и `exponential_distribution` (экспоненциальный рост); каждое для некоторого заданного диапазона. Например:

```

using my_engine = default_random_engine;           // type of engine
using my_distribution = uniform_int_distribution<>; // type of distribution

my_engine eng {};                                  // the default version of the engine
my_distribution dist {1,6};                        // distribution that maps to the ints 1..6
auto die = [&]() { return dist(eng); };             // make a generator

int x = die();                                     // roll the die: x becomes a value in
[1:6]

```

Благодаря бескомпромиссному вниманию к универсальности и производительности, один эксперт назвал компонент случайных чисел стандартной библиотеки “тем, чем хочет быть каждая библиотека случайных чисел, когда вырастет”. Однако его вряд ли можно назвать “удобным для новичков”. Операторы `using` и лямбда-выражение делают код, немного более очевидным.

Для новичков (любого уровня подготовки) полностью общий интерфейс библиотеки случайных чисел может стать серьезным препятствием. Для начала часто бывает достаточно простого генератора однородных случайных чисел. Например:

```

Rand_int rnd {1,10};           // make a random number generator for [1:10]
int x = rnd();                 // x is a number in [1:10]

```

Итак, как мы могли бы это получить? Мы должны получить что-то похожее на, `die()`, объединяющее движок и распределение внутри класса `Rand_int`:

```

class Rand_int {
public:
    Rand_int(int low, int high) :dist{low, high} { }
    int operator()() { return dist(re); }           // draw an int
    void seed(int s) { re.seed(s); }               // choose new random engine seed
private
    default_random_engine re;
    uniform_int_distribution<> dist;
};

```

Это определение по-прежнему относится к “экспертному уровню”, но с *использованием* `Rand_int()` можно справиться на первой неделе курса C++ для новичков. Например:

```
int main()
{
    constexpr int max = 9;
    Rand_int rnd {0,max};           // make a uniform random number generator

    vector<int> histogram(max+1);    // make a vector of appropriate size
    for (int i=0; i!=200; ++i)
        ++histogram[rnd()];        // fill histogram with the frequencies of numbers
    [0:max]

    for (int i = 0; i!=histogram.size(); ++i) {    // write out a bar graph
        cout << i << '\t' ;
        for (int j=0; j!=histogram[i]; ++j) cout << '*' ;
        cout << '\n' ;
    }
}
```

Результатом является (обнадеживающе скучное) равномерное распределение (с разумным статистическим разбросом):

```
0 *****
1 *****
2 *****
3 *****
4 *****
5 *****
6 *****
7 *****
8 *****
9 *****
```

Стандартной графической библиотеки для C++ не существует, поэтому я использую “ASCII графику”. Очевидно, что для C++ существует множество графических библиотек с открытым исходным кодом и коммерческих графических интерфейсов, но в этой книге я ограничусь средствами стандарта ISO.

Чтобы получить повторяющуюся или другую последовательность значений, мы указываем seed для движка; то есть мы присваиваем его внутреннему состоянию новое значение. Например:

```
Rand_int rnd {10,20};
for (int i = 0; i<10; ++i) cout << rnd() << ' ';    // 16 13 20 19 14 17 10 16 15 14
cout << '\n';
rnd.seed(999);
for (int i = 0; i<10; ++i) cout << rnd() << ' ';    // 11 17 14 19 20 13 20 14 16 19
cout << '\n';
rnd.seed(999);
for (int i = 0; i<10; ++i) cout << rnd() << ' ';    // 11 17 14 19 20 13 20 14 16 19
cout << '\n';
```

Повторяющиеся последовательности важны для детерминированной отладки. Заполнение разными значениями важно, когда мы не хотим повторения. Если вам нужны подлинные случайные числа, а не сгенерированная псевдослучайная последовательность, посмотрите, как `random_device` реализован на вашем компьютере.

17.6 Векторная арифметика

`vector`, описанный в §12.2, был разработан как общий механизм хранения значений, чтобы быть гибким и вписываться в архитектуру контейнеров, итераторов и алгоритмов. Однако он не поддерживает математические векторные операции. Добавить такие операции в `vector` было бы несложно, но его общность и гибкость исключают оптимизацию, которая часто считается необходимой для серьезной работы с числами. Следовательно, стандартная библиотека предоставляет (в `<valarray>`) похожий на `vector` шаблон, называемый `valarray`, который является менее общим и лучше поддается оптимизации для численных вычислений:

```
template<typename T>
class valarray {
    // ...
};
```

Для `valarray` поддерживаются обычные арифметические операции и наиболее распространенные математические функции. Например:

```
void f(valarray<double>& a1, valarray<double>& a2)
{
    valarray<double> a = a1*3.14+a2/a1;      // numeric array operators *, +, /, and
    =
    a2 += a1*3.14;
    a = abs(a);
    double d = a2[7];
    // ...
}
```

Операции являются векторными операциями; то есть они применяются к каждому элементу задействованных векторов.

В дополнение к арифметическим операциям `valarray` предлагает пошаговый доступ для реализации многомерных вычислений.

17.7 Числовые ограничения

В `<limits>` стандартная библиотека предоставляет классы, которые описывают свойства встроенных типов - такие как максимальный показатель степени `float` или количество байт в `int`. Например, мы можем проверить утверждение, что `char` знаковый:

```
static_assert(numeric_limits<char>::is_signed, "unsigned characters!");
static_assert(100000<numeric_limits<int>::max(), "small ints!");
```

Второе утверждение работает, только потому что `numeric_limits<int>::max()` является функцией `constexpr` (§1.6).

Мы можем определить `numeric_limits` для наших собственных пользовательских типов.

17.8 Псевдонимы типов

Размер фундаментальных типов, таких как `int` и `long long` определяется реализацией; то есть они могут отличаться в разных реализациях C++. Если нам нужно уточнить размер наших целых чисел, мы можем использовать псевдонимы, определенные в `<stdint>`, такие как `int32_t` и `uint_least64_t`. Последнее означает целое число без знака длиной не менее 64 бит.

Любопытный суффикс `_t` - это пережиток времен C, когда считалось важным, чтобы имя отражало то, что оно называет псевдоним.

Другие распространенные псевдонимы, такие как `size_t` (тип, возвращаемый оператором `sizeof`) и `ptrdiff_t` (тип результата вычитания одного указателя из другого), можно найти в `<stddef>`.

17.9 Математические константы

При выполнении математических вычислений нам нужны общие математические константы, такие как `e`, `pi` и `log2e`. Стандартная библиотека предлагает это и многое другое. Они бывают двух форм: шаблон, который позволяет нам указать точный тип (например, `pi_v<T>`) и краткое название для наиболее распространенного использования (например, `pi` означает `pi_v<double>`). Например:

```
void area(float r)
{
    using namespace std::numbers; // this is where the mathematical constants are kept

    double d = pi*r*r;
    float f = pi_v<float>*r*r;

    // ...
}
```

В этом случае разница невелика (нам пришлось бы печатать с точностью 16 знаков или около того, чтобы увидеть ее), но в реальных физических расчетах такие различия быстро становятся значительными. Другими областями, где важна точность констант, являются графика и искусственный интеллект, где все большее значение приобретают меньшие представления значений.

В `<numbers>` можно найти `e` (число Эйлера), `log2e` (\log_2 из `e`), `log10e` (\log_{10} из `e`), `pi`, `inv_pi` ($1/\pi$), `inv_sqrtpi` ($1/\sqrt{\pi}$), `ln2`, `ln10`, `sqrt2` ($\sqrt{2}$), `sqrt3` ($\sqrt{3}$), `inv_sqrt3` ($1/\sqrt{3}$), `egamma` (постоянная Эйлера-Маскерони) и `phi` (золотое сечение).

Естественно, нам хотелось бы иметь больше математических констант и констант для других областей. Это легко сделать, потому что такие константы представляют собой шаблоны переменных со специализациями для `double` (или любого другого типа, наиболее полезного для области применения):

```
template<typename T>
constexpr T tau_v = 2*pi_v<T>;

constexpr double tau = tau_v<double>;
```

17.10 Советы

- [1] Численные проблемы часто неуловимы. Если вы не уверены на 100% в математических аспектах численной задачи, либо воспользуйтесь советом эксперта, либо поэкспериментируйте, либо сделайте и то, и другое; §17.1.
- [2] Не пытайтесь выполнять серьезные числовые вычисления, используя только ядро языка; используйте библиотеки; §17.1.
- [3] Рассмотрите `accumulate()`, `inner_product()`, `partial_sum()`, и `adjacent_difference()` прежде чем писать цикл для вычисления значения из последовательности; §17.3.
- [4] Для больших объемов данных попробуйте параллельные и векторизованные алгоритмы; §17.3.1.
- [5] Используйте `std::complex` для арифметики комплексных чисел; §17.4.

- [6] Свяжите генератор и распределение, чтобы получить генератор случайных чисел; §[17.5](#).
- [7] Позаботьтесь, чтобы ваши случайные числа были достаточно случайными для использования по назначению; §[17.5](#).
- [8] Не используйте стандартную библиотеку C `rand()`; она является недостаточно случайной для реального использования; §[17.5](#).
- [9] Используйте `valarray` для числовых вычислений, когда эффективность во время выполнения важнее гибкости в отношении операций и типов элементов; §[17.6](#).
- [10] Свойства числовых типов доступны через `numeric_limits`; §[17.7](#).
- [11] Используйте `numeric_limits` чтобы проверить, что числовые типы подходят для их использования; §[17.7](#).
- [12] Используйте псевдонимы для целочисленных типов, если вы хотите быть уверенными в отношении их размеров; §[17.8](#).

Параллелизм

Делайте это просто: настолько просто, насколько это возможно, но не проще.
– A. Einstein

- [Введение](#)
- [Задачи и thread](#)

[Передача аргументов](#); [Возвращение результатов](#)

- [Обмен данными](#)

[mutex](#) и [блокировки](#); [atomic](#)

- [Ожидание событий](#)
- [Коммуникации задач](#)

[future](#) и [promise](#); [packaged_task](#); [async\(\)](#); [Остановка thread](#)

- [Корутины \(сопрограммы\)](#)

[Кооперативная многозадачность](#)

- [Советы](#)

18.1 Введение

Параллелизм – выполнение нескольких задач одновременно – широко используется для повышения пропускной способности (за счет использования нескольких процессоров для одного вычисления) или для повышения быстродействия (позволяя одной части программы выполнять работу, в то время как другая ожидает ответа). Все современные языки программирования обеспечивают поддержку этого. Поддержка, предоставляемая стандартной библиотекой C++, является переносимым и типобезопасным вариантом того, что используется в C++ уже более 20 лет и почти повсеместно поддерживается современным оборудованием. Поддержка стандартной библиотеки в первую очередь направлена на поддержку параллелизма на системном уровне, а не на прямое предоставление сложных моделей параллелизма более высокого уровня; они могут предоставляться в виде библиотек, созданных с использованием средств стандартной библиотеки.

Стандартная библиотека напрямую поддерживает одновременное выполнение нескольких потоков в одном адресном пространстве. Чтобы обеспечить это, C++ предоставляет подходящую модель памяти и набор атомарных операций. Атомарные опера-

ции позволяют программировать без блокировок [Dechev,2010]. Модель памяти гарантирует, что до тех пор, пока программист избегает состояния гонки за данные (неконтролируемый параллельный доступ к изменяемым данным), все работает так, как можно было бы наивно ожидать. Однако большинство пользователей увидят параллелизм только в терминах стандартной библиотеки и библиотек, построенных поверх нее. В этом разделе кратко приведены примеры основных средств поддержки параллелизма стандартной библиотеки: `thread`, `mutex`, операции `lock()`, `packaged_task` и `future`. Эти функции основаны непосредственно на том, что предлагают операционные системы, и не приводят к снижению производительности по сравнению с ними. Они также не гарантируют значительного повышения производительности по сравнению с тем, что предлагает операционная система.

Не рассматривайте параллелизм как панацею. Если задачу можно выполнять последовательно, то часто это делается проще и быстрее. Передача информации из одного потока в другой может оказаться на удивление дорогостоящей.

В качестве альтернативы использованию явных функций параллелизма мы часто можем использовать параллельный алгоритм для использования нескольких механизмов выполнения для повышения производительности (§13.6, §17.3.1).

Наконец, C++ поддерживает корутины (сопрограммы), то есть функции, которые сохраняют свое состояние между вызовами (§18.6).

18.2 Задачи и `thread`

Мы называем вычисление, которое потенциально может выполняться одновременно с другими вычислениями, *задачей (task)*. *Поток (thread)*- это представление задачи в программе на системном уровне. Задача, которая должна выполняться одновременно с другими задачами, запускается путем создания `thread` (находящегося в `<thread>`) с задачей в качестве аргумента. Задача - это функция или функциональный объект:

```
void f();                // function

struct F {               // function object
    void operator()();    // F's call operator (§7.3.2)
};

void user()
{
    thread t1 {f};        // f() executes in separate thread
    thread t2 {F{}};      // F{ }() executes in separate thread

    t1.join();            // wait for t1
    t2.join();            // wait for t2
}
```

Функции `join()` гарантируют, что мы не выйдем из `user()` до завершения потоков. “Присоединиться” к потоку означает “дождаться завершения потока”.

Легко забыть `join()`, и результаты этого обычно плохие, поэтому стандартная библиотека предоставляет `jthread`, который является “присоединённым `thread`”, который следует RAII вызывая `join()` с помощью своего деструктора:

```
void user()
{
    jthread t1 {f};        // f() executes in separate thread
    jthread t2 {F{}};      // F{ }() executes in separate thread
}
```

Присоединение к потоку выполняется деструкторами, поэтому порядок построения обратный. Здесь мы ждем завершения `t2` до `t1`.

Потоки программы совместно используют одно адресное пространство. В этом потоки отличаются от процессов, которые обычно напрямую не обмениваются данными. Поскольку потоки совместно используют адресное пространство, они могут взаимодействовать через общие объекты (§18.3). Такая связь обычно контролируется блокировками или другими механизмами для предотвращения состояния гонки за данные (неконтролируемый параллельный доступ к переменной).

Программирование параллельных задач может быть *очень* сложным делом. Рассмотрим возможные реализации задач `f` (функция) и `F` (функциональный объект):

```
void f()
{
    cout << "Hello ";
}

struct F {
    void operator()() { cout << "Parallel World!\n"; }
};
```

Это пример серьезной ошибки: здесь и `f`, и `F` используют объект `cout` без какой-либо формы синхронизации. Результирующий вывод был бы непредсказуемым и мог бы варьироваться при различных выполнениях программы, поскольку порядок выполнения отдельных операций в двух задачах не определен. Программа может выдавать “странный” вывод, например

```
PaHerallllel o World!
```

Только определённая в стандарте гарантия спасает нас от гонки данных в рамках определения `ostream`, которая может привести к сбою.

Чтобы избежать подобных проблем с выходными потоками, либо попросите только один `thread` использовать выходной поток, либо используйте `osyncstream` (§11.7.5)

При определении задач многопоточной программы наша цель состоит в том, чтобы полностью разделить задачи, за исключением случаев, когда они взаимодействуют простыми и очевидными способами. Самый простой способ представить параллельную задачу как функцию, которая выполняется одновременно с вызывающей ее. Чтобы это сработало, нам просто нужно передать аргументы, вернуть результат и убедиться, что между ними нет использования общих данных (никаких соревнований за данные).

18.2.1 Передача аргументов

Как правило, для работы задачи требуются данные. Мы можем легко передавать данные (или указатели, или ссылки на данные) в качестве аргументов. Рассмотрим:

```
void f(vector<double>& v);           // function: do something with v

struct F {                          // function object: do something with v
    vector<double>& v;
    F(vector<double>& vv) :v{vv} { }
    void operator()();              // application operator; §7.3.2
};

int main()
{
    vector<double> some_vec {1, 2, 3, 4, 5, 6, 7, 8, 9};
    vector<double> vec2 {10, 11, 12, 13, 14};
```

```

    jthread t1 {f, ref(some_vec)};    // f(some_vec) executes in a separate thread
    jthread t2 {F{vec2}};            // F(vec2)() executes in a separate thread
}

```

F{vec2} принимает ссылку на вектор в качестве аргумента в **F**. Теперь **F** может использовать этот вектор, и, надеюсь, никакая другая задача не обращается к **vec2** во время выполнения **F**. Передача **vec2** по значению устранила бы этот риск.

Инициализация с помощью **{f, ref(some_vec)}** использует конструктор вариадик шаблона **thread**, который может принимать произвольную последовательность аргументов (§8.4). **ref()** - это функция типа из **<functional>**, которая, к сожалению, необходима для указания вариадик шаблону обрабатывать **some_vec** как ссылку, а не как объект. Без **ref()** аргумент **some_vec** передавался бы по значению. Компилятор проверяет, что первый аргумент может быть вызван с учетом следующих аргументов, и создает необходимый объект функции для передачи потоку. Таким образом, если **F::operator()** и **f()** выполняют один и тот же алгоритм, то обработка двух задач примерно эквивалентна: в обоих случаях создается функциональный объект для запуска **thread**.

18.2.2 Возвращение результатов

В примере, приведенном в §18.2.1, я передаю аргументы по неконстантной ссылке. Я делаю это только в том случае, если ожидаю, что задача изменит значение упомянутых данных (§1.7). Это несколько хитрый, но нередкий способ вернуть результат. Более явный метод заключается в передаче входных данных по **const** ссылке и передаче указателя или ссылки на местоположение результата в качестве отдельного аргумента:

```

void f(const vector<double>& v, double* res); // take input from v; place result in
*res

class F {
public:
    F(const vector<double>& vv, double* p) :v{vv}, res{p} { }
    void operator()();                // place result in *res
private:
    const vector<double>& v;            // source of input
    double* res;                      // target for output
};

double g(const vector<double>&);        // use return value

void user(vector<double>& vec1, vector<double> vec2, vector<double> vec3)
{
    double res1;
    double res2;
    double res3;

    thread t1 {f, cref(vec1), &res1}; // f(vec1, &res1) executes in a separate thread
    thread t2 {F{vec2, &res2}};       // F{vec2, &res2}() executes in a separate thread
    thread t3 { [&](){ res3 = g(vec3); } }; // capture local variables by reference

    t1.join(); // join before using results
    t2.join();
    t3.join();

    cout << res1 << ' ' << res2 << ' ' << res3 << '\n';
}

```

Здесь **cref(vec1)** передает **const** ссылку на **vec1** в качестве аргумента для **t1**.

Это работает, и техника очень распространена, но я не считаю возврат результатов через ссылки особенно элегантным, поэтому я возвращаюсь к этой теме в §18.5.1.

18.3 Обмен данными

Иногда задачам требуется обмениваться данными. В этом случае доступ должен быть синхронизирован таким образом, чтобы одновременно имела доступ не более чем одна задача. Опытные программисты воспримут это как упрощение (например, нет проблем с одновременным чтением многими задачами неизменяемых данных), но подумайте, как обеспечить, чтобы не более чем одна задача одновременно имела доступ к заданному набору объектов.

18.3.1 `mutex` и блокировки

`mutex`, “объект взаимного исключения”, является ключевым элементом общего обмена данными между `thread`. `thread` получает `mutex` с помощью операции `lock()`:

```
mutex m;           // controlling mutex
int sh;            // shared data

void f()
{
    scoped_lock lck {m};           // acquire mutex
    sh += 7;                       // manipulate shared data
} // release mutex implicitly
```

Тип `lck` выводится как `scoped_lock<mutex>` (§7.2.3). Конструктор `scoped_lock` получает мьютекс (посредством вызова `m.lock()`). Если другой поток уже получил мьютекс, поток ожидает (“блокируется”) до тех пор, пока другой поток не завершит свой доступ. Как только поток завершает свой доступ к общим данным, `scoped_lock` освобождает `mutex` (вызовом `m.unlock()`). Когда `mutex` освобождается, `thread`, ожидающие его, возобновляют выполнение (“пробуждаются”). Средства взаимного исключения и блокировки находятся в `<mutex>`.

Обратите внимание на использование RAII (§6.3). Использование дескрипторов ресурсов, таких как `scoped_lock` и `unique_lock` (§18.4), проще и намного безопаснее, чем явная блокировка и разблокировка `mutex`.

Соответствие между общими данными и `mutex` зависит от соглашения: программист должен знать, какой `mutex` должен соответствовать каким данным. Очевидно, что это чревато ошибками, и столь же очевидно, что мы стараемся сделать соответствие понятным с помощью различных языковых средств. Например:

```
class Record {
public:
    mutex rm;
    // ...
};
```

Не нужно быть гением, чтобы догадаться, что для `Record` с именем `rec` вы должны захватить `rec.rm`, прежде чем получить доступ к остальной части `rec`, хотя комментарий или более подходящее название могли бы помочь читателю.

Нередки случаи, когда для выполнения какого-либо действия требуется одновременный доступ к нескольким ресурсам. Это может привести к тупиковой ситуации. Например, если `thread1` получает `mutex1`, а затем пытается получить `mutex2`, в то время как `thread2` получает `mutex2`, а затем пытается получить `mutex1`, то ни одна из задач никогда не

будет продолжена дальше. `scoped_lock` помогает, позволяя нам получать несколько блокировок одновременно:

```
void f()
{
    scoped_lock lck {mutex1,mutex2,mutex3};    // acquire all three locks
    // ... manipulate shared data ...

} // implicitly release all mutexes
```

Этот `scoped_lock` будет запущен только после получения всех его аргументов `mutex` и никогда не будет блокироваться (“переходить в спящий режим”) при удержании `mutex`. Деструктор для `scoped_lock` гарантирует, что `mutex` будут освобождены, когда `thread` покинет область видимости.

Обмен информацией через совместно используемые данные – довольно низкоуровневое средство. В частности, программист должен разработать способы узнать, какая задача была выполнена, а какая нет в рамках различных задач. В этом отношении использование общих данных уступает понятию вызова и возврата. С другой стороны, некоторые люди убеждены, что совместное использование должно быть более эффективным, чем копирование аргументов и возвращаемых данных. Это действительно может быть так, когда речь идет о больших объемах данных, но блокировка и разблокировка являются относительно дорогостоящими операциями. Кроме того, современные машины очень хорошо копируют данные, особенно компактные, такие как элементы `vector`. Поэтому не выбирайте общие данные для обмена информацией бездумно, из-за “эффективности” и, желательно, проводите измерения быстродействия.

Базовый `mutex` позволяет одному потоку одновременно получать доступ к данным. Один из наиболее распространенных способов обмена данными - это обмен между многими читающими и одним пишущим потоком. Эта идиома “блокировка читающий-пишущий” поддерживается `shared_mutex`. Читающий поток получит мьютекс “shared”, чтобы другие читающие все еще могли получить доступ, в то время как пишущий поток будет требовать эксклюзивного доступа. Например:

```
shared_mutex mx;    // a mutex that can be shared

void reader()
{
    shared_lock lck {mx};    // willing to share access with other readers
    // ... read ...
}

void writer()
{
    unique_lock lck {mx};    // needs exclusive (unique) access
    // ... write ...
}
```

18.3.2 atomic

`mutex` - это довольно тяжелый механизм, задействующий операционную систему. Он позволяет выполнять произвольные объемы работы без состояний гонки за данные. Однако существует гораздо более простой и дешевый механизм для выполнения лишь небольшого объема работы: `atomic` переменная. Например, вот простой вариант классической блокировки с двойной проверкой:

```
mutex mut;
atomic<bool> init_x;    // initially false.
```

```

X x;                                // variable that requires nontrivial initialization

if (!init_x) {
    lock_guard lck {mut};
    if (!init_x) {
        // ... do nontrivial initialization of x ...
        init_x = true;
    }
}

// ... use x ...

```

atomic избавляет нас от большинства вариантов использования гораздо более дорогого **mutex**. Если бы **init_x** не был **atomic**, эта инициализация завершилась бы неудачей очень редко, вызывая загадочные и трудноулавливаемые ошибки, потому что произошло бы соревнование за данные в **init_x**.

Здесь я использовал **lock_guard**, а не **scoped_lock**, потому что мне нужен был только один **mutex**, поэтому было достаточно простейшей блокировки (**lock_guard**).

18.4 Ожидание событий

Иногда **thread** необходимо дождаться какого-либо внешнего события, например, завершения другим **thread** задачи или истечения определенного промежутка времени. Самое простое “событие” - это просто течение времени. Используя средства для работы времени, найденные в **<chrono>**, я могу написать:

```

using namespace chrono;            // see §16.2.1

auto t0 = high_resolution_clock::now();
this_thread::sleep_for(milliseconds{20});
auto t1 = high_resolution_clock::now();

cout << duration_cast<nanoseconds>(t1-t0).count() << " nanoseconds passed\n";

```

Мне даже не нужно было запускать **thread**; по умолчанию **this_thread** может ссылаться на один-единственный поток.

Я использовал **duration_cast**, чтобы настроить единицы измерения часов на нужные мне наносекунды.

Базовая поддержка обмена данными с использованием внешних событий обеспечивается **condition_variable**, находящимся в **<condition_variable>**. **condition_variable** - это механизм, позволяющий одному **thread** ожидать другой **thread**. В частности, это позволяет **thread** ожидать возникновения некоторого условия (часто называемого *событием*) в результате работы, выполняемой другими **thread**.

Использование **condition_variable** поддерживает множество форм элегантного и эффективного совместного использования, но может быть довольно сложным. Рассмотрим классический пример взаимодействия двух **thread** путем передачи сообщений через **queue**. Для простоты я объявляю **queue** и механизм предотвращения условий соревнования в этой **queue** глобальными для производителя и потребителя:

```

class Message {                    // object to be communicated
    // ...
};

queue<Message> mqueue;              // the queue of messages
condition_variable mcond;          // the variable communicating events
mutex mmutex;                      // for synchronizing access to mcond

```


Типы `queue`, `condition_variable` и `mutex` предоставляются стандартной библиотекой. `consumer()` читает и обрабатывает `Message`:

```
void consumer()
{
    while(true) {
        unique_lock lck {mmutex};           // acquire mmutex
        mcond.wait(lck,[] { return !mqueue.empty(); }); // release mmutex and wait;
                                                // re-acquire mmutex upon wakeup
        auto m = mqueue.front();             // don't wake up unless mqueue is non-empty
        mqueue.pop();                        // get the message
        lck.unlock();                        // release mmutex
        // ... process m ...
    }
}
```

Здесь я явно защищаю операции с `queue` и с `condition_variable` с помощью `unique_lock` для `mutex`. Ожидание `condition_variable` освобождает его аргумент блокировки до тех пор, пока ожидание не закончится (чтобы очередь не была пустой), а затем повторно запрашивает его. Явная проверка условия, здесь `!mqueue.empty()`, защищает от пробуждения только для того, чтобы обнаружить, что какая-то другая задача “добралась туда первой”, так что условие больше не выполняется.

Я использовал `unique_lock`, а не `scoped_lock` по двум причинам:

- Нам нужно передать блокировку в `condition_variable wait()`. `scoped_lock` нельзя переместить, а `unique_lock` можно
- Мы хотим разблокировать `mutex`, защищающий переменную условия, перед обработкой сообщения. `unique_lock` предлагает операции, такие как `lock()` и `unlock()`, для низкоуровневого управления синхронизацией.

С другой стороны, `unique_lock` может обрабатывать только один `mutex`.

Соответствующий `producer` выглядит следующим образом:

```
void producer()
{
    while(true) {
        Message m;
        // ... fill the message ...
        scoped_lock lck {mmutex};           // protect operations
        mqueue.push(m);
        mcond.notify_one();                 // notify
    }                                       // release mmutex (at end of scope)
}
```

18.5 Коммуникации задач

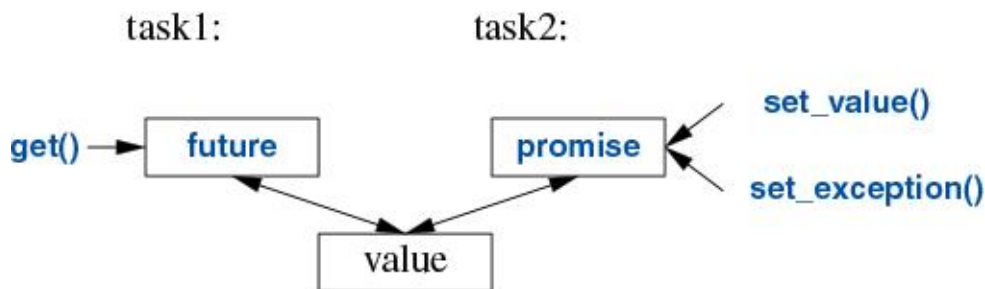
Стандартная библиотека предоставляет несколько возможностей, позволяющих программистам работать на концептуальном уровне задач (работа, которая потенциально может выполняться одновременно), а не непосредственно на более низком уровне потоков и блокировок:

- `future` и `promise` для возврата значения из задачи, созданной в отдельном потоке
- `packaged_task`, помогающий запускать задачи и подключать механизмы для возврата результата
- `async()` для запуска задачи способом, очень похожим на вызов функции

Эти объекты находятся в `<future>`.

18.5.1 `future` и `promise`

Важным моментом в отношении `future` и `promise` является то, что они позволяют передавать значение между двумя задачами без явного использования блокировки; “система” эффективно реализует передачу. Основная идея проста: когда задача хочет передать значение другой, она помещает это значение в `promise`. Каким-то образом реализация заставляет это значение отображаться в соответствующей `future`, из которой оно может быть прочитано (обычно с помощью средства запуска задачи). Мы можем представить это графически:



Если у нас есть `future<X>` с именем `fx`, мы можем `get()` из него значение типа `X`:

```
X v = fx.get();    // if necessary, wait for the value to get computed
```

Если значения еще нет, наш поток блокируется до тех пор, пока оно не поступит. Если значение не может быть вычислено, функция `get()` может бросить исключение (из системы или переданное из `promise`).

Основное назначение `promise` - обеспечить простые операции “назначения” (называемые `set_value()` и `set_exception()`) соответствующие `future` `get()`. Названия “будущее(`future`)” и “обещание(`promise`)” являются историческими; пожалуйста, не вините меня и не приписывайте мне заслуг. Они являются еще одним благодатным источником каламбуров.

Если у вас есть `promise` и вам нужно отправить результат типа `X` в `future`, вы можете сделать одну из двух вещей: передать значение или передать исключение. Например:

```
void f(promise<X>& px)    // a task: place the result in px
{
    // ...
    try {
        X res;
        // ... compute a value for res ...
        px.set_value(res);
    }
    catch (...) {          // oops: couldn't compute res
        px.set_exception(current_exception());    // pass the exception to the
                                                    // future's thread
    }
}
```

`current_exception()` ссылается на перехваченное исключение.

Чтобы справиться с исключением, переданным через `future`, вызывающий `get()` должен быть готов перехватить его где-нибудь. Например:

```
void g(future<X>& fx)      // a task: get the result from fx
{
    // ...
    try {
        X v = fx.get();    // if necessary, wait for the value to get computed
    }
```

```

        // ... use v ...
    }
    catch (...) {                // oops: someone couldn't compute v
        // ... handle error ...
    }
}

```

Если ошибка не должна обрабатываться самой **g()**, код сводится к минимальному:

```

void g(future<X>& fx)              // a task: get the result from fx
{
    // ...
    X v = fx.get();              // if necessary, wait for the value to be computed
    // ... use v ...
}

```

Теперь исключение, генерируемое функцией **fx (f())**, неявно передается вызывающему **g()**, точно так же, как это было бы, если бы **g()** вызывал **f()** напрямую.

18.5.2 packaged_task

Как нам включить **future** в задачу, которой нужен результат, и соответствующий **promise** в поток, который должен выдать этот результат? Тип **packaged_task** предусмотрен для упрощения настройки задач, связанных с **future** и **promise** для запуска в **thread**. **packaged_task** предоставляет код-обертку для помещения возвращаемого значения или исключения из задачи в **promise** (подобно коду, показанному в §18.5.1). Если вы запросите его, вызвав **get_future**, **packaged_task** выдаст вам **future**, соответствующее его **promise**. Например, мы можем настроить две задачи, чтобы каждая добавляла половину элементов **vector<double>**, используя **accumulate()** стандартной библиотеки (§17.3):

```

double accum(vector<double>::iterator beg, vector<double>::iterator end, double init)
    // compute the sum of [beg:end) starting with the initial value init
{
    return accumulate(&*beg, &*end, init);
}

double comp2(vector<double>& v)
{
    packaged_task pt0 {accum};                // package the task (i.e., accum)
    packaged_task pt1 {accum};
    future<double> f0 {pt0.get_future()};     // get hold of pt0's future
    future<double> f1 {pt1.get_future()};     // get hold of pt1's future

    double* first = &v[0];
    thread t1 {move(pt0), first, first+v.size()/2, 0}; // start a thread for pt0
    thread t2 {move(pt1), first+v.size()/2, first+v.size(), 0}; // start a thread for
pt1
    // ...

    return f0.get()+f1.get();                // get the results
}

```

Шаблон **packaged_task** принимает тип задачи в качестве аргумента шаблона (здесь **double (double*, double*, double)**) и задачу в качестве аргумента конструктора (здесь **accum**). Операции **move()** необходимы, поскольку **packaged_task** не может быть скопирован. Причина, по которой **packaged_task** не может быть скопирован, заключается в том, что это дескриптор ресурса: он владеет своим **promise** и (косвенно) несет ответственность за любые ресурсы, которыми может владеть его задача.

Пожалуйста, обратите внимание на отсутствие явного упоминания о блокировках в этом коде: мы можем сосредоточиться на задачах, которые необходимо выполнить, а не на механизмах, используемых для управления их передачей. Эти две задачи будут выполняться в отдельных потоках и, таким образом, потенциально параллельно.

18.5.3 `async()`

Подход, которого я придерживался в этой главе, который я считаю самым простым, но в то же время одним из самых действенных: рассматривайте задачу как функцию, которая может выполняться одновременно с другими задачами. Это далеко не единственная модель, поддерживаемая стандартной библиотекой C++, но она хорошо подходит для широкого спектра нужд. При необходимости можно использовать более тонкие и хитроумные модели (например, стили программирования, основанные на совместно используемой памяти).

Чтобы запустить задачи, которые потенциально могут выполняться асинхронно, мы можем использовать `async()`:

```
double comp4(vector<double>& v)
    // spawn many tasks if v is large enough
{
    if (v.size() < 10'000)                // is it worth using concurrency?
        return accum(v.begin(), v.end(), 0.0);

    auto v0 = &v[0];
    auto sz = v.size();

    auto f0 = async(accum, v0,          v0+sz/4, 0.0); // first quarter
    auto f1 = async(accum, v0+sz/4,    v0+sz/2, 0.0); // second quarter
    auto f2 = async(accum, v0+sz/2,    v0+sz*3/4, 0.0); // third quarter
    auto f3 = async(accum, v0+sz*3/4,  v0+sz,    0.0); // fourth quarter

    return f0.get()+f1.get()+f2.get()+f3.get();      // collect and combine the results
}
```

По сути, `async()` отделяет “вызываемую часть” функции от “части получения результата” и отделяет и то, и другое от фактического выполнения задачи. Используя `async()`, вам не нужно думать о потоках и блокировках. Вместо этого вы мыслите в терминах задач, которые потенциально вычисляют свои результаты асинхронно. Существует очевидное ограничение: даже не думайте использовать `async()` для задач, которые совместно используют ресурсы, нуждающиеся в блокировке. С помощью `async()` вы даже не знаете, сколько `thread` будет использоваться, потому что это зависит от `async()`, чтобы решить, основываясь на том, что он знает о системных ресурсах, доступных во время вызова. Например, `async()` может проверить, доступны ли какие-либо незанятые ядра (процессоры), прежде чем принимать решение о том, сколько `thread` использовать.

Использование предположения о стоимости вычислений относительно стоимости запуска `thread`, такого как `v.size() < 10 000`, очень примитивно и подвержено грубым ошибкам в отношении производительности. Однако здесь не подходящее место для обсуждения того, как управлять `thread`. Не воспринимайте эту оценку как нечто большее, чем простое и, вероятно, неверное предположение.

Редко возникает необходимость вручную распараллеливать алгоритм стандартной библиотеки, такой как `accumulate()`, поскольку параллельные алгоритмы (например, `reduce(par_unseq, /*...*/)`) обычно справляются с этим лучше (§17.3.1). Однако эта методика носит общий характер.

Пожалуйста, обратите внимание, что `async()` - это не просто механизм, специализирующийся на параллельных вычислениях для повышения производительности. Например, он также может быть использован для запуска задачи по получению информации от пользователя, оставляя “основную программу” активной с чем-то другим (§18.5.3).

18.5.4 Остановка `thread`

Иногда мы хотим остановить `thread`, потому что нас больше не интересует его результат. Просто “убить” его обычно неприемлемо, поскольку `thread` может владеть ресурсами, которые должны быть освобождены (например, блокировками, подпотоками и подключениями к базе данных). Вместо этого стандартная библиотека предоставляет механизм для вежливого запроса `thread` на очистку и уход: `stop_token`. `thread` может быть запрограммирован на завершение, если у него есть `stop_token` и запрошена остановка.

Рассмотрим параллельный алгоритм `find_any()`, который порождает множество `thread`, ищущих результат. Когда `thread` возвращается с ответом, мы хотели бы остановить остальные `thread`. Каждый `thread`, порожденный функцией `find_any()`, вызывает функцию `find()` для выполнения реальной работы. Эта функция `find()` является очень простым примером общего стиля задачи, где есть основной цикл, в который мы можем вставить тест на то, следует ли продолжать или остановиться:

```
atomic<int> result = -1;    // put a resulting index here

template<class T> struct Range { T* first; T* last; }; // a way of passing a range of Ts

void find(stop_token tok, const string* base, const Range<string> r, const string target)
{
    for (string* p = r.first; p!=r.last && !tok.stop_requested(); ++p)
        if (match(*p, target)) {           // match() applies some matching criteria to
                                            // the two strings
            result = p - base;              // the index of the found element
            return;
        }
}
```

Здесь `!tok.stop_requested()` проверяет, запросил ли какой-либо другой `thread` завершение этого `thread`. `stop_token` - это механизм для безопасной (без состояний гонки за данные) передачи такого запроса.

Вот тривиальная функция `find_any()`, которая порождает всего два `thread`, выполняющих функцию `find()`:

```
void find_all(vector<string>& vs, const string& key)
{
    int mid = vs.size()/2;
    string* pvs = &vs[0];

    stop_source ss1{};
    jthread t1(find, ss1.get_token(), pvs, Range{pvs,pvs+mid}, key); // first half of
vs

    stop_source ss2{};
    jthread t2(find, ss2.get_token(), pvs, Range{pvs+mid,pvs+vs.size()} , key);
    // second half of vs

    while (result == -1)
        this_thread::sleep_for(10ms);
}
```

```

ss1.request_stop(); // we have a result: stop all threads
ss2.request_stop();

    // ... use result ...
}

```

`stop_source` создает `stop_token`, через которые запросы на остановку передаются `thread`.

Синхронизация и возврат результата - это самое простое, что я мог придумать: поместите результат в `atomic` переменную (§18.3.2) и выполняйте проверку в бесконечном цикле.

Конечно, мы могли бы доработать этот простой пример, чтобы использовать множество потоков поиска, сделать возврат результатов более общим и использовать различные типы элементов. Однако это скрыло бы основную роль `stop_source` и `stop_token`.

18.6 Корутины (сопрограммы)

Корутина (сопрограмма) - это функция, которая сохраняет свое состояние между вызовами. В этом она немного похожа на функциональный объект, но сохранение и восстановление его состояния между вызовами являются неявными и полными. Рассмотрим классический пример:

```

generator<long long> fib()           // generate Fibonacci numbers
{
    long long a = 0;
    long long b = 1;
    while (a<b) {
        auto next = a+b;
        co_yield next;              // save state, return value, and wait
        a = b;
        b = next;
    }
    co_return 0;                    // a fib too far
}

void user(int max)
{
    for (int i=0; i<max; i++)
        cout << fib() << ' ';
}

```

Этот код выведет

```
1 2 3 5 8 13 ...
```

Возвращаемое `generator` значение - это место, где сопрограмма сохраняет свое состояние между вызовами. Мы могли бы, конечно, создать функциональный объект `Fib`, который работал бы таким же образом, но тогда нам пришлось бы самим поддерживать его состояние. Для больших состояний и более сложных вычислений сохранение и восстановление состояния становятся утомительными, их трудно оптимизировать и они подвержены ошибкам. По сути, сопрограмма - это функция, которая сохраняет свой `stackframe` между вызовами. `co_yield` возвращает значение и ожидает следующего вызова. Функция `co_return` возвращает значение и завершает сопрограмму.

Сопрограммы могут быть синхронными (вызывающая сторона ожидает результата) или асинхронными (вызывающая сторона выполняет какую-либо другую работу, пока не получит результат из сопрограммы). Пример с Фибоначчи, очевидно, был синхронным. Это позволяет провести некоторые приятные оптимизации. Например, хороший

оптимизатор может встроить вызовы `fib()` и развернуть цикл, оставив только последовательность вызовов `<<`, которые сами по себе могут быть оптимизированы до

```
cout << "1 2 3 5 8 13";           // fib(6)
```

Сопрограммы реализованы в виде чрезвычайно гибкой платформы, способной обслуживать широкий спектр потенциальных применений. Она разработана экспертами и для экспертов с элементами дизайна разработанного комитетом. Это прекрасно, за исключением того, что в C++20 по-прежнему отсутствуют библиотечные средства для упрощения использования. Например, `generator` (пока) не является частью стандартной библиотеки. Тем не менее, есть предложения, и поиск в Интернете поможет вам найти хорошие реализации; [\[Cppcoro\]](#) является примером.

18.6.1 Кооперативная многозадачность

В первом томе "*Искусства компьютерного программирования*" Дональд Кнут восхваляет полезность сопрограмм, но также сетует на то, что трудно привести краткие примеры, поскольку сопрограммы наиболее полезны для упрощения сложных систем. Здесь я просто приведу тривиальный пример использования примитивов, необходимых для такого рода событийно-ориентированных симуляций, которые были основной причиной раннего успеха C++. Ключевая идея состоит в том, чтобы представить систему как сеть простых задач (сoproграмм), которые совместно выполняют сложные задачи. По сути, каждая задача - это исполнитель, который выполняет небольшую часть больших усилий. Некоторые из них являются генераторами, которые генерируют потоки запросов (возможно, используя генераторы случайных чисел, возможно, передавая данные реального мира), некоторые являются частями результатов сетевых вычислений, а некоторые генерируют выходные данные. Лично я предпочитаю, чтобы задачи (сoproграммы) взаимодействовали через очереди сообщений. Один из способов организации такой системы состоит в том, чтобы каждая задача помещала себя в очередь событий, ожидая продолжения работы после получения результата. Затем планировщик выбирает следующую задачу для запуска из очереди событий, когда это необходимо. Это форма *кооперативной многозадачности*. Я позаимствовал – с благодарностью – ключевые идеи для этого у Simula [\[Dahl, 1970\]](#), чтобы они легли в основу самой первой библиотеки C++ (§19.1.2).

Основой для разработки были:

- Множество различных *сoproграмм*, которые поддерживают свое состояние между вызовами
- Форма *полиморфизма*, которая позволяет нам сохранять списки событий, содержащих различные виды сопрограмм, и вызывать их независимо от их типов.
- *Планировщик*, который выбирает следующую сопрограмму(ы) для запуска из списка(ов).

Здесь я просто покажу пару сопрограмм и выполню их поочередно. Для таких систем важно, чтобы они не занимали слишком много места. Вот почему мы не используем процессы или потоки для таких приложений. Поток требует один или два мегабайта (в основном для его стека), а сопрограмме часто требуется всего пара десятков байт. Если вам нужно выполнить много тысяч задач, это может иметь большое значение. Переключение контекста между сопрограммами также происходит намного быстрее, чем между потоками или процессами.

Во-первых, нам нужен некоторый полиморфизм во время выполнения, который позволил бы нам единообразно вызывать десятки или сотни различных видов сопрограмм:

```
struct Event_base {
    virtual void operator>() = 0;
    virtual ~Event_base() {}
};

template<class Act>
struct Event : Event_base {
    Event(const string n, Act a) : name{ n }, act{ move(a) } {}
    string name;
    Act act;
    void operator>() override { act(); }
};
```

Event просто сохраняет действие и позволяет его вызвать; Обычно это действие представляет собой сопрограмму. Я добавил **name** просто для того, чтобы проиллюстрировать, что событие обычно содержит больше информации, чем просто дескриптор сопрограммы.

Вот тривиальное применение:

```
void test()
{
    vector<Event_base*> events = {           // create a couple of Events holding
coroutines
        new Event{ "integers ", sequencer(10) },
        new Event{ "chars ", char_seq('a') }
    };

    vector order {0, 1, 1, 0, 1, 0, 1, 0, 0}; // choose some order

    for (int x : order)                       // invoke coroutines in order
        (*events[x])();

    for (auto p : events)                     // clean up
        delete p;
}
```

Пока что в этом нет ничего специфического для сопрограммы; это просто обычная объектно-ориентированная платформа для выполнения операций над набором объектов потенциально различных типов. Однако **sequencer** и **char_seq** оказались сопрограммами. Тот факт, что они поддерживают свое состояние между вызовами, имеет важное значение для реального использования таких фреймворков:

```
task sequencer(int start, int step = 1)
{
    auto value = start;
    while (true) {
        cout << "value: " << value << '\n'; // communicate a result
        co_yield 0;                          // sleep until someone resumes this coroutine
        value += step;                       // update state
    }
}
```

Мы можем видеть, что **sequencer** - это сопрограмма, потому что она использовала **co_yield** для приостановки работы между вызовами. Это подразумевает, что **task** должна быть дескриптором сопрограммы (см. ниже).

Это намеренно тривиальная сопрограмма. Все, что он делает, - это генерирует последовательность значений и выводит их. В серьезном моделировании этот вывод прямо или косвенно стал бы входными данными для какой-либо другой сопрограммы.

`char_seq` очень похож, но другого типа для реализации полиморфизма времени выполнения:

```
task char_seq(char start)
{
    auto value = start;
    while (true) {
        cout << "value: " << value << '\n';    // communicate result
        co_yield 0;
        ++value;
    }
}
```

“Магия” заключается в возвращаемом типе `task`; он сохраняет состояние сопрограммы (фактически фрейма стека функции) между вызовами и определяет значение `co_yield`. С точки зрения пользователя `task` тривиален, он просто предоставляет оператор для вызова сопрограммы:

```
struct task {
    void operator()();
    // ... implementation details ...
}
```

Если бы `task` был в библиотеке, предпочтительно в стандартной библиотеке, это было бы все, что нам нужно было знать, но это не так, поэтому вот подсказка о том, как реализовать такие типы дескрипторов сопрограмм. Тем не менее, есть предложения, и поиск в Интернете поможет вам найти хорошие реализации; библиотека [\[Cppcoro\]](#) является примером.

Мой `task` настолько минимален, насколько я мог придумать, чтобы реализовать мои ключевые примеры:

```
struct task {
    void operator()() { coro.resume(); }

    struct promise_type {                // mapping to the language features
        suspend_always initial_suspend() { return {}; }
        suspend_always final_suspend() noexcept { return {}; }    // co_return
        suspend_always yield_value(int) { return {}; }              // co_yield
        auto get_return_object() { return task{ handle_type::from_promise(*this) }; }
        void return_void() {}
        void unhandled_exception() { exit(1); }
    };

    using handle_type = coroutine_handle<promise_type>;
    task(handle_type h) : coro(h) { }    // called by get_return_object()
    handle_type coro;                    // here is the coroutine handle
};
```

Я настоятельно рекомендую вам *не* писать такой код самостоятельно, если только вы не являетесь разработчиком библиотеки, пытающимся избавить других от лишних хлопот. Если вам интересно, в Интернете есть много объяснений.

18.7 Советы

- [1] Используйте параллелизм для повышения быстродействия или пропускной способности; §18.1.
- [2] Работайте на самом высоком уровне абстракции, который вы можете себе позволить; §18.1.
- [3] Рассматривайте процессы как альтернативу потокам; §18.1.
- [4] Средства параллелизма стандартной библиотеки типобезопасны; §18.1.
- [5] Модель памяти существует для того, чтобы избавить большинство программистов от необходимости думать об уровне архитектуры компьютеров; §18.1.
- [6] Модель памяти делает память примерно такой, какой её ожидает увидеть неискушённый программист; §18.1.
- [7] Атомики допускают программирование без блокировок; §18.1.
- [8] Оставьте программирование без блокировок экспертам; §18.1.
- [9] Иногда последовательное решение проще и быстрее, чем параллельное; §18.1.
- [10] Избегайте состояний гонки; §18.1, §18.2.
- [11] Предпочитайте параллельные алгоритмы прямому использованию параллелизма; §18.1, §18.5.3.
- [12] `thread` - это типобезопасный интерфейс к системному потоку; §18.2.
- [13] Используйте `join()`, чтобы дождаться завершения `thread`; §18.2.
- [14] Предпочитательнее использовать `jthread` чем `thread`; §18.2.
- [15] Избегайте явного обмена данными, когда это возможно; §18.2.
- [16] Предпочитайте RAII явной блокировке/разблокировке; §18.3; [CG: CP.20].
- [17] Используйте `scoped_lock` для управления `mutex`; §18.3.
- [18] Используйте `scoped_lock` для получения нескольких блокировок; §18.3; [CG: CP.21].
- [19] Используйте `shared_lock` для реализации блокировок читающий-пишущий; §18.3.
- [20] Определите `mutex` вместе с данными, которые он защищает; §18.3; [CG: CP.50].
- [21] Используйте `atomic` для очень простого совместного использования; §18.3.2.
- [22] Используйте `condition_variable` для управления взаимодействием между `thread`; §18.4.
- [23] Используйте `unique_lock` (а не `scoped_lock`), когда вам нужно скопировать блокировку или требуются манипуляции с синхронизацией более низкого уровня; §18.4.
- [24] Используйте `unique_lock` (а не `scoped_lock`) с `condition_variable`; §18.4.
- [25] Не ждите без условия; §18.4; [CG: CP.42].
- [26] Сведите к минимуму время, затрачиваемое на критическую секцию; §18.4 [CG: CP.43].
- [27] Думайте в терминах параллельных задач, а не непосредственно в терминах `thread`; §18.5.
- [28] Цените простоту; §18.5.
- [29] Предпочитайте `packaged_task` и `future` прямому использованию `thread` и `mutex`; §18.5.
- [30] Возвращайте результат, используя `promise`, и получите результат из `future`; §18.5.1; [CG: CP.60].

- [31] Используйте `packaged_task` для обработки исключений, генерируемых задачами; §18.5.2.
- [32] Используйте `packaged_task` и `future` для выражения запроса к внешней службе и ожидания ее ответа; §18.5.2.
- [33] Используйте `async()` для запуска простых задач; §18.5.3; [CG: CP.61].
- [34] Используйте `stop_token` для реализации совместного завершения; §18.5.4.
- [35] Сопрограмма может быть намного меньше потока; §18.6.
- [36] Предпочтительнее использовать библиотеки поддержки сопрограмм, чем код, созданный вручную; §18.6.

История и совместимость

*Поспешай не торопясь (festina
lente).*

– Octavius, Caesar Augustus

- [История](#)

[Временная шкала](#); [Ранние годы](#); [Стандарты ISO C++](#); [Стандарты и стиль программирования](#); [Использование C++](#); [Модель C++](#)

- [Эволюция функций C++](#)

[Языковые особенности C++11](#); [Языковые особенности C++14](#); [Языковые особенности C++17](#); [Языковые особенности C++20](#); [C++11 Компоненты стандартной библиотеки](#); [C++14 Компоненты стандартной библиотеки](#); [C++17 Компоненты стандартной библиотеки](#); [C++20 Компоненты стандартной библиотеки](#); [Удаленные и устаревшие функции](#)

- [Совместимость C/C++](#)

[C и C++ родные братья](#); [Проблемы совместимости](#)

- [Библиография](#)

- [Советы](#)

19.1 История

Я изобрел C++, написал его ранние определения и создал его первую реализацию. Я выбрал и сформулировал критерии проектирования для C++, разработал его основные языковые функции, разработал или помогал разрабатывать многие ранние библиотеки и в течение 25 лет отвечал за обработку предложений по расширению в комитете по стандартам C++.

C++ был разработан, чтобы обеспечить возможности Simula для организации программ [[Dahl, 1970](#)] в сочетании с эффективностью и гибкостью C для системного программирования [[Kernighan, 1978](#)]. Язык Simula была первоначальным источником механизмов абстракции C++. Понятие класса (с производными классами и виртуальными функциями) было заимствовано из него. Однако шаблоны и исключения появились в C++ позже из других источников вдохновения.

Эволюция C++ всегда происходила в контексте его применения. Я потратил много времени на то, чтобы выслушать пользователей, узнать мнение опытных программистов и, конечно же, написать код. В частности, мои коллеги из AT&T Bell Laboratories сыграли важную роль в развитии C++ в течение первого десятилетия его существования.

Этот раздел представляет собой краткий обзор; в нем не делается попытки упомянуть все языковые функции и библиотечные компоненты. Более того, он не вдается в подробности. Для получения дополнительной информации и, в частности, имен людей, внесших свой вклад, ознакомьтесь с тремя моими докладами на конференциях ACM по истории языков программирования [[Stroustrup, 1993](#)] [[Stroustrup, 2007](#)] [[Stroustrup, 2020](#)] и моей книгой “Дизайн и эволюция C++” (известной как “D&E”). [[Stroustrup, 1994](#)]. Они подробно описывают дизайн и эволюцию C++ и документируют влияние других языков программирования. Я стараюсь поддерживать связь между стандартными удобствами и людьми, которые предложили и усовершенствовали эти удобства. C++ - это не работа безликого, анонимного комитета или предположительно всемогущего “пожизненного диктатора”; это работа многих преданных своему делу, опытных, трудолюбивых людей.

Большинство документов, подготовленных в рамках работы по стандартизации ISO C++, доступны онлайн [[WG21](#)].

19.1.1 Временная шкала

Работа, которая привела к созданию C++, началась осенью 1979 года под названием “Си с классами”. Вот упрощенная временная шкала:

- 1979 Начата работа над “Си с классами”. Первоначальный набор функций включал классы и производные классы, управление открытым/закрытым доступом, конструкторы и деструкторы, а также объявления функций с проверкой аргументов. Первая библиотека поддерживала параллельные задачи без вытеснения и генераторы случайных чисел.
- 1984 “Си с классами” был переименован в C++. К тому времени C++ обзавелся виртуальными функциями, перегрузкой функций и операторов, ссылками, а также библиотеками потоков ввода-вывода и комплексных чисел.
- 1985 Первый коммерческий релиз C++ (14 октября). Библиотека включала потоки ввода-вывода, комплексные числа и задачи (планирование без вытеснения).
- 1985 Язык программирования C++ (“TC++PL,” October 14) [[Stroustrup, 1986](#)].
- 1989 Справочное руководство по языку программирования C++ с комментариями (“the ARM”) [[Ellis, 1989](#)].
- 1991 Язык программирования C++, второе издание [[Stroustrup, 1991](#)], представление обобщенного программирования с использованием шаблонов и обработки ошибок на основе исключений, включая общую идиому управления ресурсами “Получение ресурсов - это инициализация” (RAII).
- 1997 Язык программирования C++, третье издание [[Stroustrup, 1997](#)] представлен ISO C++, включая пространства имен, `dynamic_cast` и множество усовершенствований шаблонов. В стандартную библиотеку добавлен STL-фреймворк универсальных контейнеров и алгоритмов.
- 1998 Стандарт ISO C++ [C++, 1998].
- 2002 Началась работа над пересмотренным стандартом, в просторечии получившим название C++0x.
- 2003 Была выпущена редакция стандарта ISO C++ с “исправлением ошибок”. [C++, 2011].
- 2011 Стандарт ISO C++11 [C++, 2011] предлагает единообразную инициализацию, семантику перемещения, типы, выводимые из инициализаторов (`auto`), `for` для диапазона, шаблоны с переменным количеством аргументов, лямбда-выражения, псевдонимы типов, модель памяти, подходящую для параллелизма, и многое другое. В стандартную библиотеку добавлен `thread`, блокировки, регулярные выражения, хэш-

таблицы ([unordered_map](#)), указатели управляющие ресурсами ([unique_ptr](#) и [shared_ptr](#)) и многое другое.

2013 Появились первые полные реализации C++11.

2013 Язык программирования C++, четвёртое издание представлен C++11.

2014 Стандарт ISO C++14 [[C++,2014](#)] дополняет C++11 шаблонами переменных, разделителями цифр, универсальными лямбда-выражениями и несколькими улучшениями стандартной библиотеки. Были завершены первые реализации C++14.

2015 Начат проект по основным руководящим принципам C++ [[Stroustrup,2015](#)].

2017 Стандарт ISO C++17 [[C++,2017](#)] предлагает разнообразный набор новых функций, включая гарантии порядка вычисления, структурное связывание, выражения свёртки, библиотеку файловой системы, параллельные алгоритмы, а также типы [variant](#) и [optional](#). Были завершены первые реализации C++17.

2020 Стандарт ISO C++20 [[C++,2020](#)], предлагающий [module](#), [concept](#), сопрограммы, диапазоны, форматирование в стиле [printf\(\)](#), календари и множество второстепенных функций. Были завершены первые реализации C++20.

Во время разработки C++11 был известен как C++0x. Как это нередко бывает в крупных проектах, мы были чрезмерно оптимистичны в отношении даты завершения. Ближе к концу мы пошутили, что 'x' в C++0x был шестнадцатеричным, так что C++0x стал C++0B. С другой стороны, комитет отправил C++14, C++17 и C++20 вовремя, как и основные поставщики компиляторов.

19.1.2 Ранние годы

Первоначально я разработал и внедрил этот язык, потому что хотел распределить службы ядра UNIX по многопроцессорным и локальным сетям (то, что сейчас известно как многоядерные и кластерные). Для этого мне нужно было точно указать части системы и то, как они взаимодействуют. Simula [[Dahl,1970](#)] была бы идеальной для этого, за исключением соображений производительности. Мне также нужно было иметь дело непосредственно с аппаратным обеспечением и обеспечить высокопроизводительные механизмы параллельного программирования, для которых C был бы идеальным языком, за исключением его слабой поддержки модульности и проверки типов. Результат добавления классов в стиле Simula в C (классический C; §[19.3.1](#)), “Си с классами”, использовался для крупных проектов, в которых были тщательно протестированы его возможности для написания программ, использующих минимальное время и память. В нем отсутствовала перегрузка операторов, ссылки, виртуальные функции, шаблоны, исключения и многие-многие детали [[Stroustrup,1982](#)]. Первое использование C++ за пределами исследовательской организации началось в июле 1983 года.

Название C++ (произносится как “си плюс плюс”) было придумано Риком Маскитти летом 1983 года и выбрано мной в качестве замены “Си с классами”. Название указывает на эволюционный характер изменений по сравнению с Си; “++” - это оператор инкремента в Си. Немного более короткое название “C+” является синтаксической ошибкой; оно также использовалось как название другого языка. Знатоки семантики языка Си считают, что C++ уступает ++C. Язык не назван D, потому что он был расширением C, потому что он не пытался устранить проблемы путем удаления функций, и потому что уже существовало несколько потенциальных преемников C с именем D. Еще одну интерпретацию названия C++ смотрите в приложении [[Orwell,1949](#)].

C++ был разработан в первую очередь для того, чтобы мне и моим друзьям не приходилось программировать на ассемблере, C или различных модных тогда языках вы-

сокого уровня. Его главная цель состояла в том, чтобы сделать написание хороших программ проще и приятнее для отдельного программиста. В первые годы не существовало проекта C++ на бумаге; проектирование, документирование и реализация осуществлялись одновременно. Не было ни “проекта C++”, ни “комитета по разработке C++”. На протяжении всего этого времени C++ эволюционировал, чтобы справляться с проблемами, с которыми сталкиваются пользователи, и в результате обсуждений среди моих друзей, коллег и меня самого.

Самый первый дизайн C++ включал объявления функций с проверкой типа аргумента и неявными преобразованиями, классы с различием между `public/private` интерфейсом и реализацией, производные классы, а также конструкторы и деструкторы. Я использовал макросы для обеспечения примитивной параметризации [Stroustrup,1982]. К середине 1980-х годов это было в неэкспериментальном использовании. В конце того же года я смог представить набор языковых средств, поддерживающих согласованный набор стилей программирования. Оглядываясь назад, я считаю введение конструкторов и деструкторов наиболее важным. В терминологии того времени [Stroustrup,1979]:

функция “new” создает среду выполнения для функций-членов; функция “delete” отменяет это.

Вскоре после этого “функция new” и “функция delete” были переименованы в “конструктор” и “деструктор”. Вот корень стратегий управления ресурсами в C++ (вызывающих потребность в исключениях) и ключ ко многим методам, позволяющим сделать пользовательский код коротким и понятным. Если в то время существовали другие языки, которые поддерживали несколько конструкторов, способных выполнять общий код, я о них не знал (и не хочу знать). Деструкторы были новинкой в C++.

C++ был выпущен коммерчески в октябре 1985 года. К тому времени я добавил встраивание (§1.3, §5.2.1), `const` (§1.6), перегрузку функций (§1.3), ссылки (§1.7), перегрузку операторов (§5.2.1, §6.4) и виртуальные функции (§5.4). Из этих функций поддержка полиморфизма во время выполнения в виде виртуальных функций была, безусловно, самой спорной. Я знал о его ценности из Simula, но считал невозможным убедить большинство людей в мире системного программирования в его ценности. Системные программисты, как правило, относились к косвенным вызовам функций с подозрением, а людям, знакомым с другими языками, поддерживающими объектно-ориентированное программирование, было трудно поверить, что `virtual` функции могут быть достаточно быстрыми, чтобы быть полезными в системном коде. И наоборот, многим программистам с объектно-ориентированным образованием было (и многим до сих пор приходится) трудно привыкнуть к идее, что вы используете вызовы виртуальных функций только для выражения выбора, который должен быть сделан во время выполнения. Соппротивление виртуальным функциям может быть связано с сопротивлением идее о том, что вы можете получить более совершенные системы за счет более регулярной структуры кода, поддерживаемой языком программирования. Многие программисты на Си, похоже, убеждены, что на самом деле важна полная гибкость и тщательная индивидуальная проработка каждой детали программы. Мое мнение состояло (и остается) в том, что нам нужна любая помощь, которую мы можем получить от языков и инструментов: внутренняя сложность систем, которые мы пытаемся создать, всегда находится на грани того, что мы можем выразить.

Ранние документы (например, [Stroustrup,1985] и [Stroustrup,1994]) описывали C++ следующим образом: C++ - это язык программирования общего назначения, который

- лучше чем Си
- поддерживает абстракцию данных
- поддерживает объектно-ориентированное программирование

Обратите внимание, что я *не* говорил “С++ - это объектно-ориентированный язык программирования”. Здесь “поддерживает абстракцию данных” относится к сокрытию информации, классам, которые не являются частью иерархий классов, и обобщенному программированию. Первоначально обобщённое программирование слабо поддерживалось за счет использования макросов [Stroustrup,1982]. Шаблоны и концепты появились гораздо позже.

Большая часть разработки С++ была выполнена на классных досках моими коллегами. В первые годы отзывы Стю Фельдмана, Александра Фрейзера, Стива Джонсона, Брайана Кернигана, Дуга Макилроя и Денниса Ричи были бесценны.

Во второй половине 1980-х я продолжал добавлять языковые функции в ответ на комментарии пользователей и руководствуясь своими общими целями в отношении С++. Наиболее важными из них были шаблоны [Stroustrup,1988] и обработка исключений [Koenig,1990], которые считались экспериментальными на момент начала работы над стандартами. При разработке шаблонов я был вынужден выбирать между гибкостью, эффективностью и ранней проверкой типов. В то время никто не знал, как одновременно получить все три. Чтобы конкурировать с кодом в стиле С для требовательных системных приложений, я почувствовал, что должен выбрать первые два свойства. Оглядываясь назад, я думаю, что выбор был правильным, и продолжающийся поиск лучшей проверки типов шаблонов [DosReis,2006] [Gregor,2006] [Sutton,2011] [Stroustrup,2012a] [Stroustrup,2017] привел к концептам С++20 (глава 8). Разработка исключений была сосредоточена на многоуровневом распространении исключений, передаче произвольной информации обработчику ошибок и интеграции между исключениями и управлением ресурсами путем использования локальных объектов с деструкторами для представления и освобождения ресурсов. Я неуклюже назвал этот критический метод *получение ресурсов это инициализация*, а другие вскоре сократили его до аббревиатуры *RAII* (§6.3).

Я обобщил механизмы наследования С++ для поддержки нескольких базовых классов [Stroustrup,1987]. Это называлось *множественным наследованием* и считалось сложным и спорным. Я считал это гораздо менее важным, чем шаблоны или исключения. Множественное наследование абстрактных классов (часто называемых *интерфейсами*) в настоящее время является универсальным в языках, поддерживающих статическую проверку типов и объектно-ориентированное программирование.

Язык С++ развивался рука об руку с некоторыми ключевыми библиотечными возможностями. Например, я разработал классы `complex` [Stroustrup,1984], `vector`, `stack` и `(I/O) stream` [Stroustrup,1985] вместе с механизмами перегрузки операторов. Первые классы `string` и `list` были разработаны Джонатаном Шопиро и мной в рамках одной и той же работы. Классы `string` и `list` Джонатана были первыми, которые получили широкое применение в качестве части библиотеки. Класс `string` из стандартной библиотеки С++ уходит своими корнями в эти ранние разработки. Библиотека задач, описанная в [Stroustrup,1987b], была частью первой программы “C with Classes”, написанной в 1980 году. Она предоставляла сопрограммы и планировщик. Я написал его и связанные с ним классы для поддержки симуляций в стиле `Simula`. Это имело решающее значение для успеха С++ и его широкого распространения в 1980-х годах. К сожалению, нам пришлось ждать до 2011 года (30 лет!), чтобы стандартизировать и сделать общедоступной поддержку параллелизма (§18.6). Сопрограммы являются частью С++20 (§18.6). На разработку средства создания шаблонов повлияло множество шаблонов `vector`, `map`, `list` и `sort`, разработанных Эндрю Кенигом, Алексом Степановым, мной и другими программистами.

Наиболее важным нововведением в стандартной библиотеке 1998 года был STL, фреймворк алгоритмов и контейнеров ([главы 12, 13](#)). Это была работа Алекса Степана (совместно с Дэйвом Массером, Мэн Ли и другими), основанная на более чем десятилетней работе над обобщённым программированием. STL оказал огромное влияние как в сообществе C++, так и за его пределами.

C++ вырос в среде с множеством устоявшихся и экспериментальных языков программирования (например, Ada [[Ichbiah,1979](#)], Algol 68 [[Woodward,1974](#)] и ML [[Paulson,1996](#)]). В то время я свободно владел примерно 25 языками, и их влияние на C++ описано в [[Stroustrup,1994](#)] и [[Stroustrup,2007](#)]. Однако определяющее влияние всегда оказывали приложения, с которыми я сталкивался. Моей преднамеренной политикой было то, чтобы эволюция C++ была “ориентирована на решение проблем”, а не на подражание.

19.1.3 Стандарты ISO C++

Взрывной рост использования C++ вызвал некоторые изменения. Где-то в течение 1987 года стало ясно, что формальная стандартизация C++ неизбежна и что нам нужно заложить основу для усилий по стандартизации [[Stroustrup,1994](#)]. Результатом стали сознательные усилия по поддержанию контакта между разработчиками компиляторов C++ и их основными пользователями. Это было сделано с помощью бумажной и электронной почты, а также посредством личных встреч на конференциях по C++ и в других местах.

AT&T Bell Labs внесла значительный вклад в развитие C++ и в расширение его сообщества, позволив мне поделиться проектами переработанных версий справочного руководства по C++ с разработчиками и пользователями. Поскольку многие из этих людей работали в компаниях, которые можно было бы рассматривать как конкурирующие с AT&T, значение этого вклада не следует недооценивать. Менее просвещенная компания могла бы вызвать серьезные проблемы с фрагментацией языка, просто ничего не предпринимая. Так получилось, что около сотни человек из десятков организаций прочитали и прокомментировали то, что стало общепринятым справочным руководством и базовым документом для усилий по стандартизации ANSI C++. Их имена можно найти в *аннотированном справочном руководстве по C++* (“the ARM”) [[Ellis,1989](#)]. Комитет X3J16 ANSI был создан в декабре 1989 года по инициативе Hewlett-Packard, DEC и IBM при поддержке AT&T. В июне 1991 года эта ANSI (американская национальная) стандартизация C++ стала частью усилий ISO (международной) по стандартизации C++. Комитет ISO по C++ называется WG21. Начиная с 1990 года, эти объединенные комитеты по стандартам C++ были основным форумом для эволюции C++ и уточнения его определения. Я работал в этих комитетах на протяжении всего времени. В частности, будучи председателем рабочей группы по расширениям (позже названной эволюционной группой) с 1990 по 2014 год, я непосредственно отвечал за обработку предложений по серьезным изменениям в C++ и добавлению новых языковых возможностей. Первоначальный проект стандарта для общественного обсуждения был подготовлен в апреле 1995 года. Первый стандарт ISO C++ (ISO/IEC 14882-1998) [[C++,1998](#)] был ратифицирован национальным голосованием 22:0 в 1998 году. “Выпуск с исправлением ошибок” этого стандарта был выпущен в 2003 году, поэтому иногда можно услышать, как люди ссылаются на C++03, но это, по сути, тот же язык и стандартная библиотека, что и C++98.

C++11, известный в течение многих лет как C++0x, является результатом работы членов WG21. Комитет работал в условиях все более обременительных процессов и процедур, навязываемых самим себе. Эти процессы, вероятно, привели к улучшению (и более строгой) спецификации, но они также ограничили инновации [[Stroustrup,2007](#)].

Первоначальный проект стандарта для общественного обсуждения был подготовлен в 2009 году. Второй стандарт ISO C++ (ISO/IEC 14882-2011) [[C++,2011](#)] был ратифицирован национальным голосованием 21:0 в августе 2011 года.

Одна из причин большого разрыва между двумя стандартами заключается в том, что у большинства членов комитета (включая меня) сложилось ошибочное впечатление, что правила ISO требуют “периода ожидания” после выпуска стандарта, прежде чем начинать работу над новыми функциями. Следовательно, серьезная работа над новыми языковыми возможностями началась только в 2002 году. Другие причины включали увеличение размера современных языков и их базовых библиотек. Что касается страниц текста стандарта, то язык вырос примерно на 30%, а стандартная библиотека - примерно на 100%. Во многом это увеличение было связано с более подробной спецификацией, а не с новой функциональностью. Кроме того, при работе над новым стандартом C++, очевидно, требовалось проявлять большую осторожность, чтобы не скомпрометировать старый код несовместимыми изменениями. Используются миллиарды строк кода на C++, которые комитет не должен нарушать. Стабильность на протяжении десятилетий - важнейшая “особенность”.

C++11 значительно расширил стандартную библиотеку и дополнил набор функций, необходимых для стиля программирования, представляющего собой синтез “парадигм” и идиом, которые оказались успешными в C++98.

Общими целями работы над C++11 были:

- Сделать C++ лучшим языком для системного программирования и создания библиотек.
- Упростить преподавание и освоение C++.

Цели задокументированы и подробно описаны в [[Stroustrup,2007](#)].

Были предприняты значительные усилия для того, чтобы сделать программирование параллельных систем типобезопасным и переносимым. Это включало модель памяти (§[18.1](#)) и поддержку программирования без блокировок, это была работа Ханса Бема, Брайана Макнайта и других членов рабочей группы по параллелизму. Вдобавок ко всему, мы добавили библиотеку [thread](#).

После выхода C++11 все согласились с тем, что 13 лет между стандартами - это слишком много. Херб Саттер предложил комитету принять политику доставки грузов вовремя с фиксированными интервалами - “модель поезда”. Я решительно выступал за короткий интервал между стандартами, чтобы свести к минимуму вероятность задержек, потому что кто-то настаивал на дополнительном времени, позволяющем включить “еще одну важную функцию”. Мы согласовали амбициозный график на 3 года, исходя из идеи, что нам следует чередовать второстепенные и крупные релизы.

C++14 намеренно был второстепенным релизом, направленным на “завершение C++11”. Это отражает реальность того, что при фиксированной дате выпуска появятся функции, которые, как мы знаем, нам нужны, но которые мы не сможем предоставить вовремя. Кроме того, после широкого использования неизбежно обнаружатся пробелы в наборе функций.

C++17 должен был стать большим релизом. Под “большим” я подразумеваю наличие функций, которые изменят наше представление о структуре нашего программного обеспечения и о том, как мы его разрабатываем. По этому определению, C++17 был в лучшем случае выпуском среднего уровня. Он включал в себя множество незначительных расширений, но функции, которые могли бы внести кардинальные изменения (например, концепты, модули и сопрограммы), были либо не готовы, либо погрязли в противоречиях и отсутствии направления проектирования. В результате C++17 включает в себя немного для всех, но ничего такого, что существенно изменило бы жизнь программиста на C++, который уже усвоил уроки C++11 и C++14.

C++20 предлагает давно обещанные и столь необходимые основные функции, такие как модули (§3.2.2), концепты (§8.2), сопрограммы (§18.6), диапазоны (§14.5) и множество второстепенных функций. Это такое же серьезное обновление C++, как и C++11. Он стал широко доступен в конце 2021 года.

Комитет по стандартам ISO C++, SC22/WG21, в настоящее время насчитывает около 350 членов, из которых около 250 присутствовали на последнем очном совещании перед пандемией в Праге, где C++20 был одобрен единогласно 79:0, который позже был ратифицирован национальным органом голосованием 22:0. Достижение такой степени согласия среди такой большой и разнообразной группы - тяжелая работа. Опасности включают в себя “разработку комитетом”, раздутие функций, отсутствие единообразного стиля и недальновидные решения. Добиться прогресса в создании более простого в использовании и связного языка очень сложно. Комитет осознает это и пытается противостоять этому; см. [Wong,2020]. Иногда нам это удается, но очень трудно избежать сложностей, возникающих из-за “второстепенных полезных функций”, моды и желания экспертов напрямую обслуживать редкие особые случаи.

19.1.4 Стандарты и стиль

Стандарт гласит, что и как будет работать. В нем не говорится, что представляет собой хорошее и эффективное использование. Существуют значительные различия между пониманием технических деталей функций языка программирования и эффективным их использованием в сочетании с другими функциями, библиотеками и инструментами для создания более качественного программного обеспечения. Под “лучшим” я подразумеваю “более легко обслуживаемый, менее подверженный ошибкам и более быстрый”. Нам необходимо разрабатывать, популяризировать и поддерживать согласованные стили программирования. Кроме того, мы должны поддерживать эволюцию старого кода к этим более современным, эффективным и последовательным стилям.

С развитием языка и его стандартной библиотеки проблема популяризации эффективных стилей программирования стала критической. Чрезвычайно трудно заставить большие группы программистов отказаться от чего-то, что работает, ради чего-то лучшего. Все еще есть люди, которые рассматривают C++ как несколько незначительных дополнений к Си, и люди, которые считают объектно-ориентированные стили программирования 1980-х годов, основанные на массивных иерархиях классов, вершиной разработки. Многие все еще пытаются хорошо использовать современный C++ в средах с большим количеством старого кода на C++. С другой стороны, есть также много тех, кто с энтузиазмом злоупотребляет новыми возможностями. Например, некоторые программисты убеждены, что истинным C++ является только код, использующий огромное количество шаблонного метапрограммирования.

Что такое *современный C++*? В 2015 году я решил ответить на этот вопрос, разработав набор руководящих принципов написания кода, подкрепленных четко сформулированными обоснованиями. Вскоре я обнаружил, что не одинок в решении этой проблемы, и вместе с людьми из многих уголков мира, в частности из Microsoft, Red Hat и Facebook, запустил проект “C++ Core Guidelines” [Stroustrup,2015]. Это амбициозный проект, направленный на обеспечение полной безопасности типов и ресурсов в качестве основы для более простого, быстрого, безопасного и поддерживаемого кода [Stroustrup,2015b] [Stroustrup,2021]. В дополнение к конкретным правилам написания кода с обоснованиями, мы подкрепляем рекомендации инструментами статического анализа и крошечной библиотекой поддержки. Я считаю, что нечто подобное необходимо для продвижения сообщества C++ в целом вперед, чтобы извлечь выгоду из улучшений языковых функций, библиотек и вспомогательных инструментов.

19.1.5 Использование C++

В настоящее время C++ является очень широко используемым языком программирования. Число его пользователей быстро росло с одного в 1979 году до примерно 400 000 в 1991 году; то есть число пользователей удваивалось примерно каждые 7,5 месяцев в течение более чем десяти лет. Естественно, темпы роста замедлились после этого первоначального скачка, но, по моим лучшим оценкам, в 2018 году было около 4,5 миллионов программистов на C++ [[Kazakova,2015](#)] и, возможно, на миллион больше сегодня (2022). Большая часть этого роста произошла после 2005 года, когда экспоненциальный рост скорости процессора прекратился, и важность языковой производительности возросла. Этот рост был достигнут без формального маркетинга или организованного сообщества пользователей [[Stroustrup,2020](#)].

C++ - это прежде всего промышленный язык; то есть он более заметен в промышленности, чем в образовании или изучении языков программирования. Он возник в Bell Labs, вдохновленный разнообразными и жесткими потребностями телекоммуникаций и системного программирования (включая драйверы устройств, сетевые технологии и встраиваемые системы). С этого момента использование C++ распространилось практически во всех отраслях: микроэлектронике, веб-приложениях и инфраструктуре, операционных системах, финансовой, медицинской, автомобильной, аэрокосмической, физике высоких энергий, биологии, производстве энергии, машинном обучении, видеоиграх, графике, анимации, виртуальной реальности и многом другом. В основном он используется там, где для решения задач требуется сочетание возможностей C++ эффективно использовать аппаратное обеспечение и управлять сложностью. По-видимому, это постоянно расширяющийся набор применений [[Stroustrup,1993](#)] [[Stroustrup,2014](#)] [[Stroustrup,2020](#)].

19.1.6 Модель C++

Язык C++ можно кратко охарактеризовать как набор взаимоподдерживающих средств:

- Система статических типов с равной поддержкой встроенных и пользовательских типов ([Chapter 1](#), [Chapter 5](#), [Chapter 6](#))
- Семантика значений и ссылок (§[1.7](#), §[5.2](#), §[6.2](#), [Chapter 12](#), §[15.2](#))
- Систематическое и общее управление ресурсами (RAII) (§[6.3](#))
- Поддержка эффективного объектно-ориентированного программирования (§[5.3](#), [class.virtual](#), §[5.5](#))
- Поддержка гибкого и эффективного обобщённого программирования ([Chapter 7](#), [Chapter 18](#))
- Поддержка программирования времени компиляции (§[1.6](#), [Chapter 7](#), [Chapter 8](#))
- Прямое использование ресурсов компьютера и операционной системы (§[1.4](#), [Chapter 18](#))
- Поддержка параллелизма с помощью библиотек (часто реализуемая с использованием интринсиков) ([Chapter 18](#))

Компоненты стандартной библиотеки обеспечивают дополнительную существенную поддержку для достижения этих целей высокого уровня.

19.2 Эволюция функций C++

Здесь я перечисляю языковые возможности и компоненты стандартной библиотеки, которые были добавлены в C++ для стандартов C++11, C++14, C++17 и C++20.

19.2.1 Языковые особенности C++11

Просмотр списка языковых функций может привести в замешательство. Помните, что языковая функция не предназначена для использования изолированно. В частности, большинство функций, которые являются новыми в C++11, не имеют смысла в отрыве от фреймворка, предоставляемого более старыми функциями.

- [1] Единообразная и общая инициализация с использованием `{}`-списков (§1.4.2, §5.2.3)
- [2] Вывод типа из инициализатора: `auto` (§1.4.2)
- [3] Предотвращение сужающих преобразований (§1.4.2)
- [4] Обобщенные и гарантированные постоянные выражения: `constexpr` (§1.6)
- [5] Цикл `for` для диапазона (§1.7)
- [6] Ключевое слово обозначающее нулевой указатель: `nullptr` (§1.7.1)
- [7] Ограниченный по области видимости и строго типизированный `enum`: `enum class` (§2.4)
- [8] Проверки во время компиляции: `static_assert` (§4.5.2)
- [9] Языковое отображение `{}`-списка в `std::initializer_list` (§5.2.3)
- [10] Ссылки `Rvalue`, включающие семантику перемещения (§6.2.2)
- [11] Лямбды (§7.3.3)
- [12] Вариативные шаблоны (с переменным количеством аргументов) (§7.4.1)
- [13] Псевдонимы типов и шаблонов (§7.4.2)
- [14] Символы Unicode
- [15] Целоцисленный тип `long long`
- [16] Элементы управления выравниванием: `alignas` и `alignof`
- [17] Возможность использовать тип выражения в качестве типа в объявлении: `decltype`
- [18] Сырые строковые литералы (§10.4)
- [19] Синтаксис суффиксной записи типа возвращаемого значения (§3.4.4)
- [20] Синтаксис для атрибутов и два стандартных атрибута: `[[carries_dependency]]` и `[[noreturn]]`
- [21] Способ предотвращения распространения исключений: спецификатор `noexcept` (§4.4)
- [22] Проверка возможности генерации `throw` в выражении: оператор `noexcept`
- [23] Особенности C99: расширенные целочисленные типы (т.е. правила для необязательных целочисленных типов большего размера); объединение узких/ широких строк; `__STDC_HOSTED__`; `_Pragma(X)`; макрос `vaarg` и пустые аргументы макроса
- [24] `__func__` как имя строки, содержащей имя текущей функции
- [25] `inline` пространств имён
- [26] Делегирующие конструкторы
- [27] Инициализаторы элементов внутри класса (§6.1.3)

- [28] Контроль методов по умолчанию: `default` и `delete` (§6.1.1)
- [29] Операторы явного преобразования типов
- [30] Пользовательские литералы (§6.6)
- [31] Более явный контроль над созданием экземпляра `template: extern template`
- [32] Аргументы шаблона по умолчанию для шаблонов функций
- [33] Наследуемые конструкторы (§12.2.2)
- [34] Управление преропределением: `override` (§5.5) и `final`
- [35] Более простое и общее правило SFINAE (сбой замены не является ошибкой)
- [36] Модель памяти (§18.1)
- [37] Локальное хранилище потоков: `thread_local`

Более полное описание изменений, внесенных в C++98 в C++11, смотрите в [\[Stroustrup,2013\]](#).

19.2.2 Языковые особенности C++14

- [1] Выведение типа возвращаемого значения функции; (§3.4.3)
- [2] Улучшены функции `constexpr`, например, разрешены циклы `for` (§1.6)
- [3] Шаблоны переменных (§7.4.1)
- [4] Бинарные литералы (§1.4)
- [5] Разделители цифр (§1.4)
- [6] Обобщённые лямбды (§7.3.3.1)
- [7] Более общий захват лямбд
- [8] Атрибут `[[deprecated]]`
- [9] Еще несколько незначительных расширений

19.2.3 Языковые особенности C++17

- [1] Гарантированная оптимизация копирования (§6.2.2)
- [2] Динамическое распределение выровненных типов
- [3] Более строгий порядок вычислений (§1.4.1)
- [4] UTF-8 литералы (`u8`)
- [5] Шестнадцатеричные литералы с плавающей точкой (§11.6.1)
- [6] Свертка выражений (§8.4.1)
- [7] Обобщённые значения аргументов шаблонов (`auto` параметры шаблона; §8.2.5)
- [8] Вывод типа аргумента шаблона класса (§7.2.3)
- [9] `if` времени компиляции (§7.4.3)
- [10] Операторы выбора с инициализаторами (§1.8)
- [11] `constexpr` лямбды
- [12] `inline` переменные
- [13] Структурное связывание (§3.4.5)
- [14] Новые стандартные атрибуты: `[[fallthrough]]`, `[[nodiscard]]`, и `[[maybe_unused]]`
- [15] Тип `std::byte` (§16.7)
- [16] Инициализация `enum` значением его базового типа (§2.4)
- [17] Еще несколько незначительных расширений

19.2.4 Языковые особенности C++20

- [1] Модули (§3.2.2)
- [2] Концепты (§8.2)
- [3] Сопрограммы (корутины) (§18.6)
- [4] Назначенные инициализаторы (слегка ограниченная версия функции C99)
- [5] `<=>` (“оператор космический корабль”) трехстороннее сравнение (§6.5.1)
- [6] `[*this]` чтобы захватить текущий объект по значению (§7.3.3)
- [7] Стандартные атрибуты `[[no_unique_address]]`, `[[likely]]`, и `[[unlikely]]`
- [8] Дополнительные возможности, разрешенные в функциях `constexpr`, включая `new`, `union`, `try-catch`, `dynamic_cast`, и `typeid`.
- [9] `constexpr` функции, гарантирующие вычисление во время компиляции (§1.6)
- [10] `constexpr` переменные, гарантирующие статическую инициализацию (не во время выполнения) (§1.6)
- [11] использование `using` для `enum` (§2.4)
- [12] Еще несколько незначительных расширений

19.2.5 Компоненты стандартной библиотеки C++11

Дополнения C++11 к стандартной библиотеке представлены в двух формах: новые компоненты (такие как библиотека сопоставления регулярных выражений) и улучшения компонентов C++98 (такие как конструкторы перемещения для контейнеров).

- [1] Конструкторы от `initializer_list` для контейнеров (§5.2.3)
- [2] Семантика перемещения для контейнеров (§6.2.2, §13.2)
- [3] Односвязный список: `forward_list` (§12.3)
- [4] Хэш-контейнеры: `unordered_map`, `unordered_multimap`, `unordered_set` и `unordered_multiset` (§12.6, §12.8)
- [5] Указатели управляющие ресурсами: `unique_ptr`, `shared_ptr` и `weak_ptr` (§15.2.1)
- [6] Поддержка параллелизма: `thread` (§18.2), мьютексы и блокировки (§18.3), и переменные условий (§18.4)
- [7] Высокоуровневая поддержка параллелизма: `packaged_thread`, `future`, `promise` и `async()` (§18.5)
- [8] `tuple` (§15.3.4)
- [9] Регулярные выражения: `regex` (§10.4)
- [10] Случайные числа: распределения и генераторы (§17.5)
- [11] Имена целочисленных типов, такие как `int16_t`, `uint32_t` и `int_fast64_t` (§17.8)
- [12] Контейнер непрерывной последовательности фиксированного размера: `array` (§15.3)
- [13] Копирование и повторное выбрасывание исключений (§18.5.1)
- [14] Сообщение об ошибках с использованием кодов ошибок: `system_error`
- [15] `emplace()` операции с контейнерами (§12.8)
- [16] Широкое использование `constexpr` функций
- [17] Систематическое использование `noexcept` функций
- [18] Улучшенные адаптеры функций: `function` и `bind()` (§16.3)
- [19] Преобразования `string` в числовые значения
- [20] Аллокаторы с ограниченной областью действия

- [21] Признаки типов, такие как `is_integral` и `is_base_of` (§16.4.1)
- [22] Утилиты работы с временем: `duration` и `time_point` (§16.2.1)
- [23] Рациональная арифметика времени компиляции: `ratio`
- [24] Прерывание процесса: `quick_exit` (§16.8)
- [25] Больше алгоритмов, таких как `move()`, `copy_if()` и `is_sorted()` (Chapter 13)
- [26] API для сбора мусора; позже устарел (§19.2.9)
- [27] Поддержка низкоуровневого параллелизма: `atomic` (§18.3.2)
- [28] Еще несколько незначительных расширений

19.2.6 Компоненты стандартной библиотеки C++14

- [1] `shared_mutex` и `shared_lock` (§18.3)
- [2] Пользовательские литералы (§6.6)
- [3] Адресация кортежа по типу (§15.3.4)
- [4] Гетерогенный поиск в ассоциативном контейнере
- [5] Еще несколько незначительных расширений

19.2.7 Компоненты стандартной библиотеки C++17

- [1] Файловая система (§11.9)
- [2] Параллельные алгоритмы (§13.6, §17.3.1)
- [3] Специальные математические функции (§17.2)
- [4] `string_view` (§10.3)
- [5] `any` (§15.4.3)
- [6] `variant` (§15.4.1)
- [7] `optional` (§15.4.2)
- [8] Способ вызова всего, что может быть вызвано для заданного набора аргументов: `invoke()`
- [9] Элементарные преобразования строк: `to_chars()` и `from_chars()`
- [10] Полиморфный аллокатор (§12.7)
- [11] `scoped_lock` (§18.3)
- [12] Еще несколько незначительных расширений

19.2.8 Компоненты стандартной библиотеки C++20

- [1] Диапазоны, представления и конвейеры (§14.1)
- [2] форматирование в стиле `printf()`: `format()` и `vformat()` (§11.6.2)
- [3] Календари (§16.2.2) и временные зоны (§16.2.3)
- [4] `span` для доступа на чтение и запись к непрерывным массивам (§15.2.2)
- [5] `source_location` (§16.5)
- [6] Математические константы, например `pi` и `ln10e` (§17.9)
- [7] Множество расширений для `atomic` (§18.3.2)
- [8] Способы ожидания нескольких `thread`: `barrier` и `latch`.
- [9] Макросы тестирования функций
- [10] `bit_cast<>` (§16.7)

- [11] Битовые операции (§16.7)
- [12] Добавлено больше `constexpr` функций стандартной библиотеки
- [13] Множество применений `<=>` в стандартной библиотеке
- [14] Еще много незначительных расширений

19.2.9 Удаленные и устаревшие функции

Существуют миллиарды строк C++ “где-то там”, и никто точно не знает, какие функции критичны. Следовательно, комитет ISO удаляет старые функции неохотно и после многолетних предупреждений. Однако иногда проблемные функции удаляются или *считаются устаревшими* (*deprecated*).

Объявляя устаревшей какую-либо функцию, комитет по стандартам выражает пожелание, чтобы эта функция была удалена. Однако комитет не имеет полномочий немедленно удалять активно используемую функцию, какой бы избыточной или опасной она ни была. Таким образом, “*deprecated*” - это сильный намек на то, чтобы избегать этой функции. В будущем это может исчезнуть. Список устаревших функций приведен в приложении D к стандарту [C++,2020]. Компиляторы, скорее всего, будут выдавать предупреждения об использовании устаревших функций. Однако устаревшие функции являются частью стандарта, и история показывает, что они, как правило, остаются поддерживаемыми “навсегда” по соображениям совместимости. Даже окончательно удаленные функции, как правило, продолжают существовать в реализациях из-за давления пользователей на разработчиков.

- Удалено: Спецификации исключений: `void f() throw(X,Y); // C++98; now an error`
- Удалено: Средства поддержки спецификаций исключений, `unexpected_handler`, `set_unexpected()`, `get_unexpected()` и `unexpected()`. Вместо этого используйте `noexcept` (§4.2).
- Удалено: Триграф.
- Удалено: `auto_ptr`. Вместо этого используйте `unique_ptr` (§15.2.1).
- Удалено: Использование спецификатора хранилища `register`.
- Удалено: Использование `++` на `bool`.
- Удалено: Функция `export` в C++98. Она была сложной и не поставлялась крупными поставщиками. Вместо этого `export` используется в качестве ключевого слова для модулей (§3.2.2).
- Устаревшее: Создание операций копирования для класса с деструктором (§6.2.1).
- Удалено: Присвоение строкового литерала в `char*`. Вместо этого используйте `const char*` или `auto`.
- Удалено: Некоторые функциональные объекты стандартной библиотеки C++ и связанные с ними функции. Большинство из них относятся к привязке аргументов. Вместо этого используйте лямбды и `function` (§16.3).
- Устаревшее: Сравнение значений `enum` со значениями из другого `enum` или значением с плавающей запятой.
- Устаревшее: Сравнения между двумя массивами.
- Устаревшее: Операции с запятыми в индексе (например, `[a,b]`). Чтобы освободить место для разрешения пользовательского `operator[]()` с несколькими аргументами

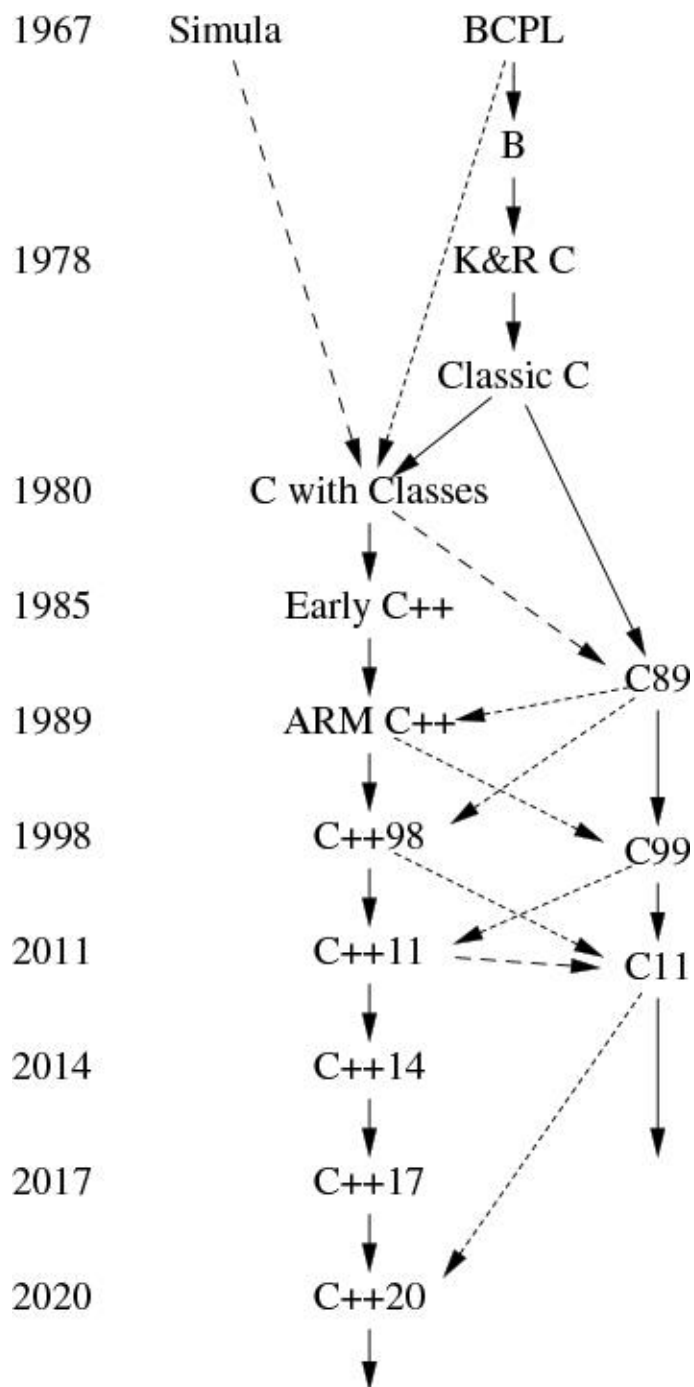
- Устаревшее: Неявный захват `*this` в лямбда-выражениях. Вместо этого используйте `[=,this]` (§7.3.3).
- Удалено: Интерфейс стандартной библиотеки для сборщиков мусора. Сборщики мусора C++ не используют этот интерфейс.
- Устаревшее: `strstream`; Вместо этого используйте `spanstream` (§11.7.4).

19.3 Совместимость C/C++

C незначительными исключениями, C++ является надмножеством Си (имеется в виду C11; [C.2011]). Большинство отличий проистекают из того, что C++ уделяет больше внимания проверке типов. Хорошо написанные программы на C, как правило, также являются программами на C++. Например, каждый пример в K&R2 [Kernighan,1988] - это C++. Компилятор может диагностировать все различия между C++ и C. Несовместимости C11/C++20 перечислены в приложении C к стандарту [C++,2020].

19.3.1 C и C++ - родные братья

Как я могу назвать C и C++ братьями и сестрами? Посмотрите на упрощенное генеалогическое древо:



Классический Си имеет двух основных потомков: ISO C и ISO C++. На протяжении многих лет эти языки развивались разными темпами и в разных направлениях. Одним из результатов этого является то, что каждый язык обеспечивает поддержку традиционного программирования в стиле Си несколько по-разному. Возникающая в результате несовместимость может сделать жизнь невыносимой для людей, использующих как C, так и C++, для людей, которые пишут на одном языке, используя библиотеки, реализованные на другом, и для разработчиков библиотек и инструментов для C и C++.

Сплошная линия означает массовое наследование функций, пунктирная линия со штрихами - заимствование основных функций, а линия точками - заимствование второстепенных функций. Из этого следует, что ISO C и ISO C++ являются двумя основными потомками K&R C [Kernighan,1978] и являются родными братьями. Каждый из них несет в себе ключевые аспекты Classic C, и ни один из них не совместим с Classic C на 100%. Я выбрал термин "Classic C" по наклейке, которая раньше была прикреплена к

терминалу Денниса Ричи. Это K&R Си плюс перечисления и присваивание `struct`. BCPL определен в [Richards,1980], а C89 - в [C1990].

Там был C++03, который я не включил в список, потому что это был релиз с исправлением ошибок. Аналогично, C17 отсутствует в списке, поскольку это версия с исправлением ошибок для C11.

Обратите внимание, что различия между C и C++ не обязательно являются результатом изменений в C, внесенных в C++. В нескольких случаях несовместимость возникает из-за несовместимых функций, внедренных в C спустя долгое время после того, как они были распространены в C++. Примерами являются возможность присвоения `T*` в `void*` и привязка глобальных `const` [Stroustrup,2002]. Иногда какая-либо функция была даже несовместимо внедрена в C после того, как она стала частью стандарта ISO C++, например, подробности о значении `inline`.

19.3.2 Проблемы совместимости

Существует множество незначительных несовместимостей между C и C++. Все это может вызвать проблемы у программиста, но в контексте C++ со всем можно справиться. Во всяком случае, фрагменты кода на C могут быть скомпилированы как C и связаны с использованием механизма `extern "C"`.

Основными проблемами при преобразовании программы на C в C++, вероятно, будут:

- Неоптимальный дизайн и стиль программирования.
- `void*` неявно преобразуется в `T*` (то есть преобразуется без приведения).
- Ключевые слова C++, такие как `class` и `private`, используемые в качестве идентификаторов в коде C.
- Несовместимая связь фрагментов кода, скомпилированных как C, и фрагментов, скомпилированных как C++.

19.3.2.1 Проблемы стиля

Естественно, программа на C написана в стиле C, таком как стиль, используемый в K & R [Kernighan,1988]. Это подразумевает широкое использование указателей и массивов и, вероятно, множества макросов. Эти средства трудно надежно использовать в большой программе. Управление ресурсами и обработка ошибок часто являются специальными (а не поддерживаемыми языком и инструментами) и часто не полностью документированы и соблюдаются. Простое построчное преобразование программы на C в программу на C++ дает программу, которая часто проверяется немного лучше. На самом деле, я никогда не преобразовывал программу на C в C++, не обнаружив каких-либо ошибок. Однако фундаментальная структура остается неизменной, как и фундаментальные источники ошибок. Если у вас была неполная обработка ошибок, утечки ресурсов или переполнение буфера в исходной программе на C, они все равно будут присутствовать в версии на C++. Чтобы получить основные преимущества, вы должны внести изменения в фундаментальную структуру кода:

- [1] Не думайте о C++ как о C с добавлением нескольких функций. C++ можно использовать таким образом, но только неоптимально. Чтобы получить действительно серьезные преимущества от C++ по сравнению с C, вам нужно применять разные стили проектирования и реализации.

- [2] Используйте стандартную библиотеку C++ в качестве преподавателя новых техник и стилей программирования. Обратите внимание на отличие от стандартной библиотеки C (например, `=` вместо `strcpy()` для копирования).
- [3] Подстановка макросов почти никогда не требуется в C++. Используйте `const` (§1.6), `constexpr` (§1.6), `enum` или `enum class` (§2.4) для определения констант, `constexpr` (§1.6), `constexpr` (§1.6) и `inline` (§5.2.1), чтобы избежать накладных расходов при вызове функций, `template` (глава 7) для указания семейств функции и типов, а также `namespace` (§3.3), чтобы избежать конфликтов имен.
- [4] Не объявляйте переменную до того, как она вам понадобится, и немедленно инициализируйте ее. Объявление может встречаться везде, где может использоваться оператор (§1.8), например, в инициализаторах оператора `for` и в условиях (§1.8).
- [5] Не используйте `malloc()`. Оператор `new` (§5.2.2) выполняет ту же работу лучше, и вместо `realloc()` попробуйте использовать `vector` (§6.3, §12.2). Не просто замените `malloc()` и `free()` на “голые” `new` и `delete` (§5.2.2).
- [6] Избегайте `void*`, `union` и приведений типов, за исключением случаев, когда это происходит глубоко внутри реализации какой-либо функции или класса. Их использование ограничивает поддержку, которую вы можете получить от системы типов, и может снизить производительность. В большинстве случаев приведение является признаком ошибки проектирования.
- [7] Если вы должны использовать явное преобразование типов, используйте соответствующее именованное приведение (например, `static_cast`; §5.2.3) для более точного описания того, что вы пытаетесь сделать.
- [8] Сведите к минимуму использование массивов и строк в стиле C. `string` стандартной библиотеки C++ (§10.2), `array` (§15.3.1) и `vector` (§12.2) часто можно использовать для написания более простого и удобного в обслуживании кода по сравнению с традиционным стилем C. В общем, старайтесь не создавать самостоятельно то, что уже было предоставлено стандартной библиотекой.
- [9] Избегайте арифметики указателей, за исключением случаев использования в очень специализированном коде (например, в менеджере памяти).
- [10] Передавайте непрерывные последовательности (например, массивы) в виде `span` (§15.2.2). Это хороший способ избежать ошибок диапазона (“переполнения буфера”) без дополнительных тестов.
- [11] Для простого обхода массива используйте `for` для диапазонов (§1.7). Это проще в написании, так же быстро, как и безопаснее, чем традиционный цикл C.
- [12] Используйте `nullptr` (§1.7.1) а не `0` или `NULL`.
- [13] Не думайте, что что-то тщательно написанное в стиле C (избегающее таких функций C++, как классы, шаблоны и исключения) более эффективно, чем более короткая альтернатива (например, использование средств стандартной библиотеки). Часто (но, конечно, не всегда) верно обратное.

19.3.2.2 `void*`

В Си `void*` может использоваться в качестве правого операнда присваивания или инициализации переменной любого типа указателя; в C++ это может быть не так. Например:

```
void f(int n)
{
    int* p = malloc(n*sizeof(int)); /* not C++; in C++, allocate using "new" */
}
```

```

    // ...
}

```

Это, вероятно, самая сложная несовместимость, с которой приходится иметь дело. Обратите внимание, что неявное преобразование `void*` в другой тип указателя в целом *небезопасно*:

```

char ch;
void* pv = &ch;
int* pi = pv;           // not C++
*pi = 666;              // overwrite ch and other bytes near ch

```

На обоих языках приведите результат `malloc()` к нужному типу. Если вы используете только C++, избегайте `malloc()`.

19.3.2.3 Линковка

C и C++ могут быть реализованы (и часто реализуются) для использования различных соглашений о связывании. Самой основной причиной этого является больший акцент C++ на проверке типов. Практическая причина заключается в том, что C++ поддерживает перегрузку, поэтому могут существовать две глобальные функции, называемые `open()`. Это должно быть отражено в том, как работает компоновщик.

Чтобы связать функцию C++ с C (чтобы ее можно было вызывать из фрагмента программы на C) или разрешить вызов функции C из фрагмента программы на C++, объявите ее `extern "C"`. Например:

```
extern "C" double sqrt(double);
```

Теперь `sqrt(double)` можно вызвать из фрагмента кода C или C++. Определение `sqrt(double)` также может быть скомпилировано как функция C или как функция C++.

Только одна функция с заданным именем в области видимости может иметь связь с C (поскольку C не допускает перегрузки функций). Спецификация привязки не влияет на проверку типов, поэтому правила C++ для вызовов функций и проверки аргументов по-прежнему применяются к функции, объявленной `extern "C"`.

19.4 Библиография

- [Boost] *The Boost Libraries: free peer-reviewed portable C++ source libraries.* www.boost.org.
- [C,1990] X3 Secretariat: *Standard – The C Language*. X3J11/90-013. ISO Standard ISO/IEC 9899-1990. Computer and Business Equipment Manufacturers Association. Washington, DC.
- [C,1999] ISO/IEC 9899. *Standard – The C Language*. X3J11/90-013-1999.
- [C,2011] ISO/IEC 9899. *Standard – The C Language*. X3J11/90-013-2011.
- [C++,1998] ISO/IEC JTC1/SC22/WG21 (editor: Andrew Koenig): *International Standard – The C++ Language*. ISO/IEC 14882:1998.
- [C++,2004] ISO/IEC JTC1/SC22/WG21 (editor: Lois Goldthwaite): *Technical Report on C++ Performance*. ISO/IEC TR 18015:2004(E) ISO/IEC 29124:2010.
- [C++,2011] ISO/IEC JTC1/SC22/WG21 (editor: Pete Becker): *International Standard – The C++ Language*. ISO/IEC 14882:2011.
- [C++,2014] ISO/IEC JTC1/SC22/WG21 (editor: Stefanus Du Toit): *International Standard – The C++ Language*. ISO/IEC 14882:2014.

- [C++,2017] ISO/IEC JTC1/SC22/WG21 (editor: Richard Smith): *International Standard – The C++ Language*. ISO/IEC 14882:2017.
- [C++,2020] ISO/IEC JTC1/SC22/WG21 (editor: Richard Smith): *International Standard – The C++ Language*. ISO/IEC 14882:2020.
- [Cppcoro] CppCoro – A coroutine library for C++. github.com/lewissbaker/cppcoro.
- [Cppreference] Online source for C++ language and standard library facilities. www.cppreference.com.
- [Cox,2007] Russ Cox: *Regular Expression Matching Can Be Simple And Fast*. January 2007. swtch.com/~rsc/regexp/regexp1.xhtml.
- [Dahl,1970] O.-J. Dahl, B. Myrhaug, and K. Nygaard: *SIMULA Common Base Language*. Norwegian Computing Center S-22. Oslo, Norway. 1970.
- [Dechev,2010] D. Dechev, P. Pirkelbauer, and B. Stroustrup: *Understanding and Effectively Preventing the ABA Problem in Descriptor-based Lock-free Designs*. 13th IEEE Computer Society ISORC 2010 Symposium. May 2010.
- [DosReis,2006] Gabriel Dos Reis and Bjarne Stroustrup: *Specifying C++ Concepts*. POPL06. January 2006.
- [Ellis,1989] Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley. Reading, Massachusetts. 1990. ISBN 0-201-51459-1.
- [Garcia,2015] J. Daniel Garcia and B. Stroustrup: *Improving performance and maintainability through refactoring in C++11*. [Isocpp.org](http://isocpp.org). August 2015. http://www.stroustrup.com/improving_garcia_stroustrup_2015.pdf.
- [Friedl,1997] Jeffrey E. F. Friedl: *Mastering Regular Expressions*. O'Reilly Media. Sebastopol, California. 1997. ISBN 978-1565922570.
- [Gregor,2006] Douglas Gregor et al.: *Concepts: Linguistic Support for Generic Programming in C++*. OOPSLA'06.
- [Ichbiah,1979] Jean D. Ichbiah et al.: *Rationale for the Design of the ADA Programming Language*. SIGPLAN Notices. Vol. 14, No. 6. June 1979.
- [Kazakova,2015] Anastasia Kazakova: *Infographic: C/C++ facts*. <https://blog.jetbrains.com/clion/2015/07/infographics-cpp-facts-before-clion/> July 2015.
- [Kernighan,1978] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language*. Prentice Hall. Englewood Cliffs, New Jersey. 1978.
- [Kernighan,1988] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language, Second Edition*. Prentice-Hall. Englewood Cliffs, New Jersey. 1988. ISBN 0-13-110362-8.
- [Knuth,1968] Donald E. Knuth: *The Art of Computer Programming*. Addison-Wesley. Reading, Massachusetts. 1968.
- [Koenig,1990] A. R. Koenig and B. Stroustrup: *Exception Handling for C++ (revised)*. Proc USENIX C++ Conference. April 1990.
- [Maddock,2009] John Maddock: *Boost.Regex*. www.boost.org. 2009. 2017.
- [Orwell,1949] George Orwell: *1984*. Secker and Warburg. London. 1949.
- [Paulson,1996] Larry C. Paulson: *ML for the Working Programmer*. Cambridge University Press. Cambridge. 1996. ISBN 978-0521565431.
- [Richards,1980] Martin Richards and Colin Whitby-Stevens: *BCPL – The Language and Its Compiler*. Cambridge University Press. Cambridge. 1980. ISBN 0-521-21965-5.
- [Stepanov,1994] Alexander Stepanov and Meng Lee: *The Standard Template Library*. HP Labs Technical Report HPL-94-34 (R. 1). 1994.
- [Stepanov,2009] Alexander Stepanov and Paul McJones: *Elements of Programming*. Addison-Wesley. Boston, Massachusetts. 2009. ISBN 978-0-321-63537-2.

- [Stroustrup,1979] Personal lab notes.
- [Stroustrup,1982] B. Stroustrup: *Classes: An Abstract Data Type Facility for the C Language*. Sigplan Notices. January 1982. The first public description of “C with Classes.”
- [Stroustrup,1984] B. Stroustrup: *Operator Overloading in C++*. Proc. IFIP WG2.4 Conference on System Implementation Languages: Experience & Assessment. September 1984.
- [Stroustrup,1985] B. Stroustrup: *An Extensible I/O Facility for C++*. Proc. Summer 1985 USENIX Conference.
- [Stroustrup,1986] B. Stroustrup: *The C++ Programming Language*. Addison-Wesley. Reading, Massachusetts. 1986. ISBN 0-201-12078-X.
- [Stroustrup,1987] B. Stroustrup: *Multiple Inheritance for C++*. Proc. EUUG Spring Conference. May 1987.
- [Stroustrup,1987b] B. Stroustrup and J. Shopiro: *A Set of C Classes for Co-Routine Style Programming*. Proc. USENIX C++ Conference. Santa Fe, New Mexico. November 1987.
- [Stroustrup,1988] B. Stroustrup: *Parameterized Types for C++*. Proc. USENIX C++ Conference, Denver, Colorado. 1988.
- [Stroustrup,1991] B. Stroustrup: *The C++ Programming Language (Second Edition)*. Addison-Wesley. Reading, Massachusetts. 1991. ISBN 0-201-53992-6.
- [Stroustrup,1993] B. Stroustrup: *A History of C++: 1979–1991*. Proc. ACM History of Programming Languages Conference (HOPL-2). ACM Sigplan Notices. Vol 28, No 3. 1993.
- [Stroustrup,1994] B. Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. Reading, Massachusetts. 1994. ISBN 0-201-54330-3.
- [Stroustrup,1997] B. Stroustrup: *The C++ Programming Language, Third Edition*. Addison-Wesley. Reading, Massachusetts. 1997. ISBN 0-201-88954-4. Hardcover (“Special”) Edition. 2000. ISBN 0-201-70073-5.
- [Stroustrup,2002] B. Stroustrup: *C and C++: Siblings, C and C++: A Case for Compatibility, and C and C++: Case Studies in Compatibility*. The C/C++ Users Journal. July-September 2002. www.stroustrup.com/papers.shtml.
- [Stroustrup,2007] B. Stroustrup: *Evolving a language in and for the real world: C++ 1991-2006*. ACM HOPL-III. June 2007.
- [Stroustrup,2009] B. Stroustrup: *Programming – Principles and Practice Using C++*. Addison-Wesley. Boston, Massachusetts. 2009. ISBN 0-321-54372-6.
- [Stroustrup,2010] B. Stroustrup: “New” Value Terminology. <https://www.stroustrup.com/terminology.pdf>. April 2010.
- [Stroustrup,2012a] B. Stroustrup and A. Sutton: *A Concept Design for the STL*. WG21 Technical Report N3351==12-0041. January 2012.
- [Stroustrup,2012b] B. Stroustrup: *Software Development for Infrastructure*. Computer. January 2012. doi:10.1109/MC.2011.353.
- [Stroustrup,2013] B. Stroustrup: *The C++ Programming Language (Fourth Edition)*. Addison-Wesley. Boston, Massachusetts. 2013. ISBN 0-321-56384-0.
- [Stroustrup,2014] B. Stroustrup: C++ Applications. <http://www.stroustrup.com/applications.shtml>.
- [Stroustrup,2015] B. Stroustrup and H. Sutter: *C++ Core Guidelines*. <https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md>.
- [Stroustrup,2015b] B. Stroustrup, H. Sutter, and G. Dos Reis: *A brief introduction to C++’s model for type- and resource-safety*. [Isocpp.org](http://www.stroustrup.com/resource-model.pdf). October 2015. Revised December 2015. <http://www.stroustrup.com/resource-model.pdf>.

- [Stroustrup,2017] B. Stroustrup: *Concepts: The Future of Generic Programming (or How to design good concepts and use them well)*. WG21 P0557R1. https://www.stroustrup.com/good_concepts.pdf. January 2017.
- [Stroustrup,2020] B. Stroustrup: *Thriving in a crowded and changing world: C++ 2006-2020*. ACM/SIGPLAN History of Programming Languages conference, HOPL-IV. June 2020.
- [Stroustrup,2021] B. Stroustrup: *Type-and-resource safety in modern C++*. WG21 P2410R0. July 2021.
- [Stroustrup,2021b] B. Stroustrup: *Minimal module support for the standard library*. P2412r0. July 2021.
- [Sutton,2011] A. Sutton and B. Stroustrup: *Design of Concept Libraries for C++*. Proc. SLE 2011 (International Conference on Software Language Engineering). July 2011.
- [WG21] ISO SC22/WG21 The C++ Programming Language Standards Committee: *Document Archive*. www.open-std.org/jtc1/sc22/wg21.
- [Williams,2012] Anthony Williams: *C++ Concurrency in Action – Practical Multithreading*. Manning Publications Co. ISBN 978-1933988771.
- [Wong,2020] Michael Wong, Howard Hinnant, Roger Orr, Bjarne Stroustrup, Daveed Vandevoorde: *Direction for ISO C++*. WG21 P2000R1. July 2020.
- [Woodward,1974] P. M. Woodward and S. G. Bond: *Algol 68-R Users Guide*. Her Majesty's Stationery Office. London. 1974.

19.5 Советы

- [1] Стандарт ISO C++ [C++,2020] определяет C++.
- [2] При выборе стиля для нового проекта или при модернизации кодовой базы полагайтесь на C++ Core Guidelines; §19.1.4.
- [3] Изучая C++, не заикливайтесь только на особенностях языка; §19.2.1.
- [4] Не заикливайтесь на устаревших десятилетиями наборах языковых функций и методах проектирования; §19.1.4.
- [5] Прежде чем использовать новую функцию в производственном коде, опробуйте ее, написав небольшие программы для проверки соответствия реализаций, которые вы планируете использовать требованиям стандартов и производительности.
- [6] Для изучения C++ используйте самую современную и полную реализацию стандарта C++, к которой вы можете получить доступ.
- [7] Общее подмножество C и C++ - не лучшее начальное подмножество C++ для изучения; §19.3.2.1.
- [8] Избегайте приведений типов; §19.3.2.1; [CG: ES.48].
- [9] Предпочитайте именованные приведения типов, такие как `static_cast`, а не приведения в стиле Си; §5.2.3; [CG: ES.49].
- [10] При преобразовании программы на C в C++ переименуйте переменные, которые являются ключевыми словами C++; §19.3.2.
- [11] Для переносимости и безопасности типов, если вам необходимо использовать C, пишите в общем подмножестве C и C++; §19.3.2.1; [CG: CPL.2].

- [12] При преобразовании программы на Си в С++ приведите результат `malloc()` к соответствующему типу или измените все виды использования `malloc()` на использование `new`; §19.3.2.2.
- [13] При преобразовании из `malloc()` и `free()` в `new` и `delete` рассмотрите возможность использования `vector`, `push_back()` и `reserve()` вместо `realloc()`; §19.3.2.1.
- [14] В С++ нет неявных преобразований из `int` в перечисления; при необходимости используйте явное преобразование типов.
- [15] Для каждого стандартного заголовка Си `<X.h>`, который помещает имена в глобальное пространство имен, заголовок `<cX>` помещает имена в пространство имен `std`.
- [16] Используйте `extern "C"` при объявлении функций; §19.3.2.3.
- [17] Предпочтительнее использование `string`, чем строки в стиле Си (прямое манипулирование массивами `char` с терминирующим нулем); [CG: SL.str.1].
- [18] Предпочитайте `iostream` Си-шному `stdio`; [CG: SL.io.3].
- [19] Предпочитайте контейнеры (например, `vector`) встроенным массивам.

Модуль `std`

*В изобретении это очень важно:
у вас должна быть целая си-
стема, которая работает.
– J. Presper Eckert*

- [Введение](#)
- [Используйте то, что предлагает Ваша реализация](#)
- [Используйте заголовки](#)
- [Сделайте свой собственный `module std`](#)
- [Советы](#)

А.1 Введение

На момент написания статьи `module std` [Stroustrup,2021b], к сожалению, еще не является частью стандарта. У меня есть основания надеяться, что это станет частью C++23. В этом приложении предлагаются некоторые идеи о том, как управлять модулями на данный момент.

Идея `module std` заключается в том, чтобы сделать все компоненты стандартной библиотеки просто и дешево доступными с помощью одного оператора `import std`; Я опирался на это на протяжении всех глав. Заголовки упоминаются и поименованы главным образом потому, что они традиционны и общедоступны, а отчасти потому, что они отражают (несовершенную) историческую организацию стандартной библиотеки.

Несколько компонентов стандартной библиотеки сбрасывают имена, такие как `sqrt()` из `<cmath>`, в глобальное пространство имен. Модуль `std` этого не делает, но когда нам нужно получить такие глобальные имена, мы можем `import std.compat`. Единственная действительно веская причина для импорта `std.compat` вместо `std` - это избежать возни со старыми базами кода, при этом все еще получая некоторые преимущества от увеличения скорости компиляции из модулей.

Обратите внимание, что модули, скорее всего, намеренно не экспортируют макросы. Если вам нужны макросы, используйте `#include`. Модули и заголовочные файлы сосуществуют; то есть, если вы оба `#include` и `import` идентичный набор объявлений, вы получите согласованную программу. Это важно для эволюции больших баз кода от использования заголовочных файлов к использованию модулей.

A.2 Используйте то, что предлагает Ваша реализация

Если немного повезет, реализация, которую мы хотим использовать, уже имеет `module std`. В этом случае нашим первым выбором должно быть его использование. Он может быть помечен как “экспериментальный”, и для его использования может потребоваться небольшая настройка или несколько опций компилятора. Итак, прежде всего, выясните, есть ли в реализации `module std` или его эквивалент. Например, в настоящее время (весна 2022) Visual Studio предлагает ряд “экспериментальных” модулей, поэтому, используя эту реализацию, мы можем определить модуль `std` следующим образом:

```
export module std;
export import std.regex;           // <regex>
export import std.filesystem;      // <filesystem>
export import std.memory;          // <memory>
export import std.threading;       // <atomic>, <condition_variable>, <future>,
                                   // <mutex>,
                                   // <shared_mutex>, <thread>
export import std.core;            // all the rest
```

Чтобы это сработало, мы, очевидно, должны использовать компилятор C++20, а также установить параметры для получения доступа к экспериментальным модулям. Имейте в виду, что все “экспериментальное” со временем изменится.

A.3 Используйте заголовки

Если реализация еще не поддерживает модули или еще не предлагает `module std` или его эквивалент, мы можем вернуться к использованию традиционных заголовков. Они являются стандартными и универсально доступными. Загвоздка в том, что для того, чтобы заставить пример работать, нам нужно выяснить, какие заголовки необходимы, и `#include` их. [Глава 9](#) может помочь здесь, и мы можем посмотреть название функции, которую мы хотим использовать, в [\[Cppreference\]](#), чтобы увидеть, частью какого заголовка она является. Если это становится утомительным, мы можем собрать часто используемые заголовки в стандартный `std.h`:

```
// std.h
#include <iostream>
#include <string>
#include <vector>
#include <list>
#include <memory>
#include <algorithms>
// ...
```

и затем

```
#include "std.h"
```

Проблема здесь в том, что `#include` такого большого количества может привести к очень медленной компиляции [\[Stroustrup,2021b\]](#).

A.4 Сделайте свой собственный `module std`

Это наименее привлекательная альтернатива, потому что, скорее всего, это потребует больше всего работы, но как только кто-то это сделает, этим можно поделиться:

```

module;
#include <iostream>
#include<string>
#include<vector>
#include<list>
#include<memory>
#include<algorithms>

```

```
// ...
```

```

export module std;
export istream;
export ostream;
export iostream;
// ...

```

Есть короткий путь:

```

export module std;

export import "iostream";
export import "string";
export import "vector";
export import "list";
export import "memory";
export import "algorithms";
// ...

```

Конструкция

```
import "iostream";
```

Импорт *заголовочного файла* - это промежуточный этап между модулями и файлами заголовков. Он берет заголовочный файл и превращает его во что-то вроде модуля, но он также может вводить имена в глобальное пространство имен (например, `#include`) и приводит к утечке макросов.

Это не так медленно компилируется, как `#include`, но и не так быстро, как правильно сконструированный именованный модуль.

A.5 Советы

- [1] Предпочитайте модули, предоставляемые реализацией; §A.2.
- [2] Используйте модули; §A.3.
- [3] Предпочитайте именованные модули блокам заголовков; §A.4.
- [4] Чтобы использовать макросы и глобальные имена из стандарта C, `import std.compat`; §A.1.
- [5] Избегайте макросов; §A.1. [CG: ES.30] [CG: ES.31].

Index

*Знание бывает двух видов. Мы
сами разбираемся в предмете
или знаем, где можем найти ин-
формацию о нём.
– Samuel Johnson*

Простите, но индекс переводчик не осилил.

Token

!=

container [169](#)

not-equal operator [7](#)

"

literal operator [85](#)

string literal [3](#)

\$, regex [131](#)

%

modulus operator [7](#)

remainder operator [7](#)

%, operator [7](#)

&

address-of operator [11](#)

pointer [11](#)

reference to [12](#)

&&, rvalue reference [77](#)

,

char [8](#)

digit separator [6](#)

(, regex [131](#)

(), call operator [94](#)

(?: pattern [134](#)

), regex [131](#)

*

contents-of operator [11](#)

iterator [179](#)

multiply operator [7](#)

operator [220](#)

pointer [11](#)

regex [131](#)

*, scaling operator [7](#)

*? lazy [132](#)

+

iterator [192](#)

plus operator [7](#)

regex [131](#)

string concatenation [125](#)

++

increment operator [7](#)

iterator [179](#), [192](#)

+=

iterator [192](#)

operator [7](#)

string append [126](#)

+? lazy [132](#)

-

iterator [192](#)

minus operator [7](#)

--

decrement operator [7](#)

iterator [192](#)

-=, iterator [192](#)

->

member access [23](#)

operator [220](#)

return type [40](#)

.

member access [23](#)

regex [131](#)

..., variadic template [114](#)

/, divide operator [7](#)

// comment [2](#)

/=, scaling operator [7](#)

: public [61](#)

<<

output operator [3](#), [84](#)

output ostream [138](#)

<=

container [169](#)

less-than-or-equal operator [7](#)

<=>

container [169](#)

spaceship operator [81](#)

<

- container [169](#)
- less-than operator [7](#)
- =
 - 0 [60](#)
 - and == [7](#)
 - assignment [17](#)
 - auto [8](#)
 - container [169](#)
 - default [56](#)
 - initializer [7](#)
 - initializer narrowing [8](#)
 - string assignment [126](#)
- ==
 - = and [7](#)
 - container [169](#)
 - equal operator [7](#)
 - iterator [192](#)
 - string [126](#)
- >
 - container [169](#)
 - greater-than operator [7](#)
- >=
 - container [169](#)
 - greater-than-or-equal operator [7](#)
- >>
 - input [istream](#) [139](#)
 - input operator [84](#)
- ?, regex [131](#)
- ?: operator [82](#)
- ?? lazy [132](#)
- [, regex [131](#)
- [&] [95](#)
- [=] [95](#)
- [[]] attribute syntax [263](#)
- []
 - array [11](#)
 - array [203](#)
 - auto [41](#)

- iterator [192](#)
- string [126](#)
- subscript operator [25](#)
- subscripting [169](#)
- \, backslash [3](#)
-], regex [131](#)
- ^, regex [131](#)
- {, regex [131](#)
- { }
 - format() argument [144](#)
 - grouping [2](#)
 - initializer [8](#)
- { }? lazy [132](#)
- |
 - pipeline [188](#)
 - regex [131](#)
- }, regex [131](#)
- ~, destructor [57](#)
- 0
 - = [60](#)
 - nullptr NULL [14](#)

A

- :%A, format() [146](#)
- abort() [225](#)
- abs() [228](#)
- abstract class [60](#)
- access
 - ., member [23](#)
 - >, member [23](#)
- accumulate() [229](#)
- acquisition RAI, resource [197](#)
- Ada [208](#)
- adaptor
 - function [216](#)
 - lambda as [216](#)
 - range [187](#)
- address, memory [16](#)

address-of operator & [11](#)

`adjacent_difference()` [229](#)

aims, C++11 [261](#)

Alexander Fraser [259](#)

algorithm [173](#)

 container [175](#)

 lifting [113](#)

 numerical [229](#)

 parallel [183](#)

 standard library [181](#)

`<algorithm>` [123](#), [182](#)

alias

 template [100](#)

 type [234](#)

 using [100](#)

`alignas` [263](#)

`alignof` [263](#)

allocation [57](#)

allocator `new`, container [167](#)

almost container [201](#)

`alnum`, regex [133](#)

`alpha`, regex [133](#)

`[[:alpha:]]` letter [133](#)

alternatives, error handling [47](#)

Annemarie [127](#)

ANSI C++ [260](#)

`any` [211](#)

Anya [208](#)

`append` `+=`, string [126](#)

argument

`{}`, `format()` [144](#)

 constrained [89](#)

 constrained template [90](#)

 default function [38](#)

 default template [108](#)

 function [37](#)

 lambda as [96](#)

 order, `format()` [145](#)

 passing, function [72](#)

 type [90](#)

 value [90](#)

arithmetic

 conversions, usual [7](#)

 operator [7](#)

 vector [233](#)

Arithmetic example [108](#), [219](#)

ARM [260](#)

array

`[]` [11](#)

 array vs. [203](#)

array [202](#)

`[]` [203](#)

`data()` [203](#)

 initialize [202](#)

`size()` [203](#)

 vs. array [203](#)

 vs. vector [203](#)

`<array>` [123](#)

`asin()` [228](#)

assembler [257](#)

`assert()`, assertion [49](#)

assertion

`assert()` [49](#)

`expect()` [48](#)

`static_assert` [50](#)

`assignable_from`, concept [190](#)

assignment

`=` [17](#)

`=`, string [126](#)

 copy [72](#), [75](#)

 initialization and [18](#)

 move [72](#), [78](#)

assignment-to-string-literal, removed [267](#)

associate type [222](#)

associative array – see [map](#)
async() launch [247](#)
at() [161](#)
atan() [228](#)
atan2() [228](#)
atexit() [225](#)
atomic [243](#)
AT&T Bell Laboratories [260](#)
attribute
 [[carries_dependency]] [263](#)
 [[deprecated]] [264](#)
 [[fallthrough]] [264](#)
 [[likely]] [265](#)
 [[maybe_unused]] [264](#)
 [[nodiscard]] [98](#), [264](#)
 [[noreturn]] [263](#)
 [[no_unique_address]] [265](#)
 syntax, [[]] [263](#)
 [[unlikely]] [265](#)
auto
 [] [41](#)
 = [8](#)
 concept and [110](#)
 return type [40](#)
auto_ptr, removed [267](#)

B

:%B, format() [146](#)
:b, format() [145](#)
back_inserter() [175](#)
backslash \ [3](#)
bad_variant_access [210](#)
base
 and derived class [61](#)
 destructor for [65](#)
basic_string [128](#)
BCPL [269](#)
begin() [83](#), [163](#), [169](#), [175](#)

beginner, book for [1](#)
Bell Laboratories, AT&T [260](#)
beta() [228](#)
bibliography [271](#)
bidirectional_iterator, concept [192](#)
bidirectional_range, concept [193](#)
binary search [182](#)
binding, structured [41](#)
bit manipulation [224](#)
bit_cast [224](#)
bit-field, bitset and [204](#)

bitset [204](#)
 and bit-field [204](#)
 and enum [204](#)
blank, regex [133](#)
block
 as function body, try [161](#)
 try [44](#)
body, function [2](#)
book for beginner [1](#)
bool [6](#)
Boolean, concept [191](#)
bounded_range, concept [193](#)
break [15](#)
Brian Kernighan [259](#)
buffer overrun [200](#)
built-in type [21](#)
byte, std::byte [224](#)

C

C [257](#)
 and C++ compatibility [268](#)
 Classic [269](#)
 difference from [268](#)
 K&R [269](#)
 style [269](#)

- with Classes [256](#)
- with Classes language features [258](#)
- with Classes standard library [259](#)
- C++
 - ANSI [260](#)
 - compatibility, C and [268](#)
 - Core Guidelines [262](#)
 - core language [2](#)
 - evolution [256](#)
 - history [255](#)
 - ISO [260](#)
 - meaning [257](#)
 - model [262](#)
 - modern [262](#)
 - pronunciation [257](#)
 - standard, ISO [2](#)
 - standard library [2](#)
 - standardization [260](#)
 - style [269](#)
 - timeline [256](#)
 - use [262](#)
 - users, number of [262](#)
- C++03 [260](#)
- C++0x, C++11 [257](#), [260](#)
- C++[11](#)
 - aims [261](#)
 - C++0x [257](#), [260](#)
 - language features [263](#)
 - library components [265](#)
- C++14 [261](#)
 - language features [264](#)
 - library components [266](#)
- C++17 [261](#)
 - language features [264](#)
 - library components [266](#)
- C++20 [1](#), [185](#), [261](#)
 - concept [104](#)
 - language features [265](#)
 - library components [266](#)
 - module [33](#)
- C++98 [260](#)
 - standard library [259](#)
- C11 [268](#)
- C+23, spanstream [149](#)
- C89 and C99 [268](#)
- C99, C89 and [268](#)
- calendar [214](#)
- call operator () [94](#)
- callback [217](#)
- capacity() [159](#), [169](#)
- capture list [95](#)
- [[carries_dependency]] attribute [263](#)
- cast [59](#)
- catch
 - clause [44](#)
 - every exception [161](#)
- catch(...) [161](#)
- cbegin() [83](#)
- ceil() [228](#)
- cend() [83](#)
- cerr [138](#)
- char [6](#)
 - ' [8](#)
- character sets, multiple [128](#)
- check
 - compile-time [50](#)
 - run-time [48](#)
- Checked_iter example [174](#)
- checking
 - cost of range [162](#)
 - template definition [109](#)
- chrono, namespace [214](#)
- <chrono> [123](#), [214](#), [243](#)
- cin [139](#)
- class [23](#), [54](#)
 - abstract [60](#)

- and `struct` [25](#)
- base and derived [61](#)
- concrete [54](#)
- hierarchy [63](#)
- interface [23](#)
- member [23](#)
- scope [9](#)
- template [88](#)
- Classic C [269](#)
- clause, requires [105](#)

- `clear()`, `iostream` [141](#)
- C-library header [123](#)
- clock [214](#)
- clock timing [243](#)
- `<cmath>` [123](#), [228](#)
- `cntrl`, `regex` [133](#)
- code complexity, function and [5](#)
- comment, `//` [2](#)
- `common_reference_with`, concept [190](#)
- `common_type_t` [190](#), [221](#)
- `common_view` [187](#)
- `common_with`, concept [190](#)
- communication, task [245](#)
- comparison operator [7](#), [81](#)
- compatibility, C and C++ [268](#)
- compilation
 - model, template [117](#)
 - separate [30](#)
- compiler [2](#)
- compile-time
 - check [50](#)
 - computation [218](#)
 - evaluation [10](#)
 - if [101](#)
- complex [55](#), [230](#)
- `<complex>` [123](#), [228](#), [230](#)
- complexity, function and code [5](#)
- components
 - C++11 library [265](#)
 - C++14 library [266](#)
 - C++17 library [266](#)
 - C++20 library [266](#)
- computation, compile-time [218](#)
- concatenation `+`, `string` [125](#)
- concept [89](#)
 - `assignable_from` [190](#)
 - `bidirectional_iterator` [192](#)
 - `bidirectional_range` [193](#)
 - Boolean [191](#)
 - `bounded_range` [193](#)
 - `common_reference_with` [190](#)
 - `common_with` [190](#)
 - `constructible_from` [191](#)
 - `contiguous_iterator` [192](#)
 - `contiguous_range` [193](#)
 - `convertible_to` [190](#)
 - `copy_constructible` [191](#)
 - `default_initializable` [191](#)
 - `derived_from` [190](#)
 - `destructible` [191](#)
 - `equality_comparable` [190](#)
 - `equality_comparable_with` [190](#)
 - `equivalence_relation` [191](#)
 - `floating_point` [190](#)
 - `forward_iterator` [192](#)
 - `forward_range` [193](#)
 - `input_iterator` [192](#)
 - `input_or_output_iterator` [192](#)
 - `input_range` [193](#)
 - `integral` [190](#)
 - `invocable` [191](#)
 - `mergeable` [192](#)
 - `mopyable` [191](#)
 - `movable` [191](#)
 - `move_constructible` [191](#)

- output_iterator [192](#)
- output_range [193](#)
- permutable [192](#)
- predicate [191](#)
- random_access_iterator [192](#)
- random_access_range [193](#)
- range [185](#)
- range [193](#)
- regular [191](#)
- regular_invocable [191](#)
- relation [191](#)
- same_as [190](#)
- semiregular [191](#)
- sentinel_for [192](#)
- signed_integral [190](#)
- sized_range [193](#)
- sized_sentinel_for [192](#)
- sortable [192](#)
- strict_weak_order [191](#)
- swappable [190](#)
- swappable_with [190](#)
- three_way_comparable [190](#)
- three_way_comparable_with [190](#)
- totally_ordered [190](#)
- totally_ordered_with [190](#)
- unsigned_integral [190](#)
- view [193](#)
- concept [104](#)
 - and auto [110](#)
 - and type [111](#)
 - and variable [111](#)
 - based overloading [106](#)
 - C++20 [104](#)
 - definition of [107](#)
 - in <concepts> [190](#)
 - in <iterator> [190](#)
 - in <ranges> [190](#)
 - static_assert and [108](#)
 - use [104](#)
- concepts
 - iterator [192](#)
 - range [193](#)
 - type [190](#)
- <concepts> [123](#)
 - concept in [190](#)
- concrete
 - class [54](#)
 - type [54](#)
- concurrency [237](#)
- condition, declaration in [67](#)
- condition_variable [244](#)
 - notify_one() [245](#)
 - wait() [244](#)
- <condition_variable> [244](#)
- const
 - immutability [10](#)
 - member function [56](#)
- constant
 - expression [10](#)
 - mathematical [234](#)
- const_cast [59](#)
- constexpr, immutability [10](#)
- constexpr
 - function [10](#)
 - if [101](#)
 - immutability [10](#)
- const_iterator [179](#)
- constrained
 - argument [89](#)
 - template [90](#)
 - template argument [90](#)
- constructible_from, concept [191](#)
- construction, order of [67](#)
- constructor [24](#)
 - and destructor [258](#)

- copy [72, 75](#)
- default [56](#)
- delegating [264](#)
- explicit [73](#)
- inherited [161](#)
- inheriting [264](#)
- initializer-list [58](#)
- invariant and [45](#)
- move [72, 77](#)
- consumer() example, producer() [244](#)
- container [57, 88, 157](#)
 - [>=](#) [169](#)
 - [==](#) [169](#)
 - [>](#) [169](#)
 - [=](#) [169](#)
 - [<](#) [169](#)
 - [<=](#) [169](#)
 - [<=>](#) [169](#)
 - [!=](#) [169](#)
 - algorithm [175](#)
 - allocator new [167](#)
 - almost [201](#)
 - object in [160](#)
 - operation [83](#)
 - overview [168](#)
 - return [176](#)
 - specialized [201](#)
 - standard library [168](#)
- contents-of operator * [11](#)
- contiguous_iterator, concept [192](#)
- contiguous_range, concept [193](#)
- conventional operation [81](#)
- conversion [73](#)
 - explicit type [59](#)
 - narrowing [8](#)
- conversions, usual arithmetic [7](#)
- convertible_to, concept [190](#)
- cooperative multitasking example [251](#)

- copy [74](#)
 - assignment [72, 75](#)
 - constructor [72, 75](#)
 - cost of [76](#)
 - elision [40, 72](#)
 - elision [78](#)
 - memberwise [72](#)
- copy() [182](#)
- copy_constructible, concept [191](#)
- copy_if() [182](#)
- Core Guidelines, C++ [262](#)
- core language, C++ [2](#)
- co_return [250](#)
- coroutine [250, 259](#)
 - generator [250](#)
 - promise_type [253](#)
- cos() [228](#)
- cosh() [228](#)
- cost
 - of copy [76](#)
 - of range checking [162](#)
- count() [182](#)
- count_if() [181–182](#)
- Courtney [208](#)
- cout [138](#)
 - output [3](#)
- co_yield [250](#)
- <cstdlib> [123](#)
- C-style
 - error handling [228](#)
 - I/O [149](#)
 - string [13](#)
- CTAD [93](#)

D

- :d, format() [145](#)
- \d, regex [133](#)
- \D, regex [133](#)

- d, regex [133](#)
- dangling pointer [196](#)
- data
 - member [23](#)
 - race [239](#)
- data(), array [203](#)
- D&E [256](#)

- deadlock [242](#)
- deallocation [57](#)
- debugging template [113](#)
- declaration [6](#)
 - function [4](#)
 - in condition [67](#)
 - interface [29](#)
 - using [36](#)
- declarator operator [12](#)
- decltype [263](#)
- decltype() [218](#)
- decrement operator -- [7](#)
- deduction
 - guide [210](#)
 - guide [92](#)
 - return type [40](#)
- default
 - = [56](#)
 - constructor [56](#)
 - function argument [38](#)
 - member initializer [74](#)
 - operation [72](#)
 - template argument [108](#)
- =default [72](#)
- defaultfloat [143](#)
- default_initializable, concept [191](#)
- definition
 - checking, template [109](#)
 - implementation [30](#)
 - of concept [107](#)

- delegating constructor [264](#)
- =delete [73](#)
- delete
 - naked [58](#)
 - operator [57](#)
- delete[], operator [57](#)
- Dennis Ritchie [259](#)
- deprecated
 - feature [267](#)
 - sstream [148, 267](#)
- [[deprecated]] attribute [264](#)
- deque [168](#)
- derived class, base and [61](#)
- derived_from, concept [190](#)
- destructible, concept [191](#)
- destruction, order of [67](#)
- destructor [57, 72](#)
 - ~ [57](#)
 - constructor and [258](#)
 - for base [65](#)
 - for member [65](#)
 - virtual [65](#)
- dictionary – see [map](#)
- difference from C [268](#)
- digit
 - [:digit:] [133](#)
 - separator ' [6](#)
- digit, regex [133](#)
- [:digit:] digit [133](#)
- directive, using [36](#)
- directory_iterator [151–152](#)
- distribution, random [231](#)
- divide operator / [7](#)
- domain error [228](#)
- double [6](#)
- double-checked locking [243](#)
- Doug McIlroy [259](#)
- drop_view [187](#)

duck typing [117](#)

duration [214](#)

duration_cast [214](#)

dynamic memory [57](#)

dynamic_cast [67](#)

is instance of [67](#)

is kind of [67](#)

E

e [234](#)

EDOM macro [228](#)

element requirements [160](#)

elision, copy [40](#), [72](#)

emplace_back() [169](#)

empty() [169](#)

enable_if [221](#)

enable_if_t [221](#)

encapsulation [72](#)

end() [83](#), [163](#), [169](#), [175](#)

endl [154](#)

engine, random [231](#)

enum

bitset and [204](#)

class enumeration [26](#)

enumeration [25](#)

using [26](#)

enumeration

enum [25](#)

enum class [26](#)

equal operator == [7](#)

equality preserving [192](#)

Equality_comparable example [108](#)

equality_comparable, concept [190](#)

equality_comparable_with, concept [190](#)

equal_range() [182](#)

equivalence_relation, concept [191](#)

ERANGE macro [228](#)

erase() [163](#), [169](#)

errno [228](#)

error

domain [228](#)

handling [43](#)

handling alternatives [47](#)

handling, C-style [228](#)

range [200](#), [228](#)

recovery [47](#)

run-time [44](#)

error-code, exception vs [47](#)

error_code [153](#)

essential operation [72](#)

evaluation

compile-time [10](#)

order of [8](#)

event driven simulation example [251](#)

evolution, C++ [256](#)

Example, expect() [48](#)

example

Arithmetic [108](#), [219](#)

Checked_iter [174](#)

cooperative multitasking [251](#)

Equality_comparable [108](#)

event driven simulation [251](#)

finally() [98](#)

find_all() [176](#)

Hello, World! [2](#)

Number [108](#)

producer() consumer() [244](#)

Rand_int [231](#)

Sentinel [193](#)

Sequence [109](#)

sum() [104](#)

task [253](#)

tau [235](#)

Value_type [109](#)

Vec [161](#)

- Vector [22–23](#), [29](#), [33–34](#), [57–58](#), [73](#), [75](#), [77](#), [88–89](#), [91–92](#), [100](#)
- exception [44](#)
 - and `main()` [161](#)
 - `catch` every [161](#)
 - specification, removed [267](#)
 - vs error-code [47](#)
- `exclusive_scan()` [229](#)
- execution policy [183](#)
- `exists()` [150](#)
- exit, program [225](#)
- `exit()` termination [225](#)
- `exp()` [228](#)
- `exp2()` [228](#)
- `expect()`
 - assertion [48](#)
 - Example [48](#)
- explicit type conversion [59](#)
- explicit constructor [73](#)
- `exponential_distribution` [231](#)
- export
 - module [33](#)
 - removed [267](#)
- expression
 - constant [10](#)
 - fold [115](#)
 - lambda [95](#)
 - requires [106](#)
- `extension()` [152](#)
- extern template [264](#)

F

- `fabs()` [228](#)
- facilities, standard library [120](#)
- `[[fallthrough]]` attribute [264](#)
- feature
 - deprecated [267](#)
 - removed [267](#)

- features
 - C with Classes language [258](#)
 - C++11 language [263](#)
 - C++14 language [264](#)
 - C++17 language [264](#)
 - C++20 language [265](#)
- file
 - header [31](#)
 - open a [151](#)
 - system operation [153](#)
 - type [154](#)
- `file_name()`, `source_location` [222](#)
- `<filesystem>` [150](#)
- `filesystem_error` [153](#)
- `filter()` [189](#)
- `filter_view` [186](#)
- final [264](#)
- `Final_action` [98](#)
- `finally()` example [98](#)
- `find()` [175](#), [182](#)
- `find_all()` example [176](#)
- `find_if()` [181–182](#)
- fixed [143](#)
- floating-point literal [6](#)
- `floating_point`, concept [190](#)
- `floor()` [228](#)
- fold expression [115](#)
- for
 - statement [12](#)
 - statement, range [12](#)
- format, output [143–144](#)
- `format()`
 - `:%A` [146](#)
 - argument `{}` [144](#)
 - argument order [145](#)
 - `:%B` [146](#)
 - `:b` [145](#)
 - `:d` [145](#)

- [:o](#) [145](#)
- precision [145](#)
- [:x](#) [145](#)
-
- [<format>](#) [123](#), [144](#)
- [forward\(\)](#) [116](#), [223](#)
- forwarding, perfect [224](#)
- [forward_iterator](#), concept [192](#)
- [forward_list](#) [168](#)
 - singly-linked list [164](#)
- [<forward_list>](#) [123](#)
- [forward_range](#), concept [193](#)
- Fraser, Alexander [259](#)
- free store [57](#)
- friend [193](#)
- [<fstream>](#) [123](#), [147](#)
- [__func__](#) [264](#)
- function [2](#)
 - adaptor [216](#)
 - and code complexity [5](#)
 - argument [37](#)
 - argument, default [38](#)
 - argument passing [72](#)
 - body [2](#)
 - body, try block as [161](#)
 - [const](#) member [56](#)
 - [constexpr](#) [10](#)
 - declaration [4](#)
 - implementation of [virtual](#) [62](#)
 - mathematical [228](#)
 - member [23](#)
 - name [5](#)
 - object [94](#)
 - overloading [5](#)
 - [return](#) value [37](#)
 - template [93](#)
 - type [217](#)
 - value [return](#) [72](#)

- function [217](#)
 - and [nullptr](#) [217](#)
- [<functional>](#) [123](#)
- [function_name\(\)](#), [source_location](#) [222](#)
- fundamental type [6](#)
- future
 - and [promise](#) [245](#)
 - member [get\(\)](#) [245](#)
- [<future>](#) [123](#), [245](#)

G

- garbage collection [79](#)
- Gavin [208](#)
- [gcd\(\)](#) [229](#)
- generator
 - coroutine [250](#)
 - type [221](#)
- generic programming [103](#), [112](#), [258](#)
- [get<>\(\)](#)
 - by index [207](#)
 - by type [207](#)
- [get\(\)](#), [future](#) member [245](#)
- [getline\(\)](#) [140](#)
- [graph](#), [regex](#) [133](#)
- greater-than operator [>](#) [7](#)
- greater-than-or-equal operator [>=](#) [7](#)
- greedy match [132](#), [135](#)
- grouping, [{}](#) [2](#)
- guide, deduction [92](#)
- Guidelines, C++ Core [262](#)

H

- half-open sequence [182](#)
- handle [24](#), [58](#)
 - resource [75](#), [198](#)
- hardware, mapping to [16](#)
- hash table [165](#)

[hash<>, unordered_map](#) [84](#)

header

 C-library [123](#)

 file [31](#)

 problems [32](#)

 standard library [121](#), [123](#)

 unit [279](#)

heap [57](#)

Hello, World! example [2](#)

hexfloat [143](#)

hierarchy

 class [63](#)

 navigation [67](#)

history, C++ [255](#)

HOPL [256](#)

I

if

 compile-time [101](#)

 constexpr [101](#)

 statement [14](#)

ifstream [147](#)

immutability

 const [10](#)

 constexpr [10](#)

 constexpr [10](#)

implementation

 definition [30](#)

 inheritance [66](#)

 iterator [178](#)

 of virtual function [62](#)

 push_back() [159](#)

 string [127](#)

import [3](#)

 and #include [277](#)

 #include and [34](#)

 module [33](#)

in-class member initialization [264](#)

#include [3](#), [31](#)

 and import [34](#)

 import and [277](#)

inclusive_scan() [229](#)

incompatibility, void* [270](#)

increment operator ++ [7](#)

index, get<>() by [207](#)

infinite range [185](#)

inheritance [61](#)

 implementation [66](#)

 interface [65](#)

 multiple [259](#)

inherited constructor [161](#)

inheriting constructor [264](#)

initialization

 and assignment [18](#)

 in-class member [264](#)

initialize [58](#)

 array [202](#)

initializer

 = [7](#)

 {} [8](#)

 default member [74](#)

 lambda as [97-98](#)

 narrowing, = [8](#)

initializer-list constructor [58](#)

initializer_list [58](#)

inline [55](#)

 namespace [264](#)

inlining [55](#)

inner_product() [229](#)

input

 istream >> [139](#)

 of user-defined type [141](#)

 operator >> [84](#)

 string [140](#)

input_iterator, concept [192](#)

`input_or_output_iterator`, concept [192](#)
`input_range`, concept [193](#)
`insert()` [163](#), [169](#)
instantiation [89](#)
instantiation time, template [117](#)
instruction, machine [16](#)
`int` [6](#)
 output bits of [204](#)
`int32_t` [234](#)
integer literal [6](#)
`integral`, concept [190](#)
interface
 class [23](#)
 declaration [29](#)
 inheritance [65](#)
invalidation [159](#)
invariant [45](#)
 and constructor [45](#)
invocable, concept [191](#)
`invoke_result_t` [221](#)
I/O [138](#)
 C-style [149](#)
 iterator and [179](#)
 state [141](#)
`<iomanip>` [143](#)
`<ios>` [123](#), [143](#)
`iostream`
 `clear()` [141](#)
 kinds of [146](#)
 `setstate()` [141](#)
 `unset()` [141](#)
`<iostream>` [3](#), [123](#)
`iota()` [229](#)
`is`
 instance of, `dynamic_cast` [67](#)
 kind of, `dynamic_cast` [67](#)
`is_arithmetic_v` [218](#)
`is_base_of_v` [218](#)

`is_constructible_v` [218](#)
`is_directory()` [151](#)
`is_integral_v` [218](#)
ISO
 C++ [260](#)
 C++ standard [2](#)
ISO-14882 [260](#)
`is_same_of_v` [218](#)
`istream` [138](#)
 `>>`, input [139](#)
`<istream>` [139](#)
`istream_iterator` [179](#)
`istreamstring` [147](#)
iterator [83–84](#), [175](#)
 `==` [192](#)
 `+` [192](#)
 `--` [192](#)
 `+=` [192](#)
 `-=` [192](#)
 `-` [192](#)
 `++` [179](#), [192](#)
 `*` [179](#)
 `[]` [192](#)
 and I/O [179](#)
 concepts [192](#)
 implementation [178](#)
iterator [163](#), [179](#)
`<iterator>`, concept in [190](#)
`iterator_t` [109](#)
`iter_value_t` [109](#)

J

`join()`, thread [238](#)
`join_view` [187](#)

K

Kernighan, Brian [259](#)
key and value [164](#)
kinds of `iostream` [146](#)
K&R C [269](#)

L

`\l`, regex [133](#)

`\L`, regex [133](#)

lambda

as adaptor [216](#)

as argument [96](#)

as initializer [97–98](#)

expression [95](#)

language

and library [119](#)

features, C with Classes [258](#)

features, C++11 [263](#)

features, C++14 [264](#)

features, C++17 [264](#)

features, C++20 [265](#)

launch, `async()` [247](#)

lazy

`+`? [132](#)

`{}`? [132](#)

`??` [132](#)

`*`? [132](#)

match [132](#), [135](#)

`lcm()` [229](#)

leak, resource [67](#), [78](#), [197](#)

less-than operator `<` [7](#)

less-than-or-equal operator `<=` [7](#)

letter, `[[:alpha:]]` [133](#)

library

algorithm, standard [181](#)

C with Classes standard [259](#)

C++98 standard [259](#)

components, C++11 [265](#)

components, C++14 [266](#)

components, C++17 [266](#)

components, C++20 [266](#)

container, standard [168](#)

facilities, standard [120](#)

language and [119](#)

non-standard [119](#)

standard [119](#)

lifetime, scope and [9](#)

lifting algorithm [113](#)

`[[likely]]` attribute [265](#)

`<limits>` [217](#), [234](#)

`line()`, `source_location` [222](#)

linker [2](#)

list

capture [95](#)

`forward_list` singly-linked [164](#)

`list` [162](#), [168](#)

literal

`"`, string [3](#)

floating-point [6](#)

integer [6](#)

operator `"` [85](#)

raw string [130](#)

suffix, `s` [127](#)

suffix, `sv` [129](#)

type of string [127](#)

UDL, user-defined [84](#)

user-defined [264](#)

literals

`string_literals` [127](#)

`string_view_literals` [129](#)

`ln10` [234](#)

`ln2` [234](#)

local scope [9](#)

lock, reader-writer [242](#)

locking, double-checked [243](#)

`log()` [228](#)

`log10()` [228](#)

[log2\(\)](#) [228](#)
[log2e](#) [234](#)
[long long](#) [263](#)
[lower, regex](#) [133](#)

M

machine instruction [16](#)
macro
 EDOM [228](#)
 ERANGE [228](#)
[main\(\)](#) [2](#)
 exception and [161](#)
[make_pair\(\)](#) [207](#)
[make_shared\(\)](#) [199](#)
[make_unique\(\)](#) [199](#)
management, resource [78](#), [197](#)
manipulation, bit [224](#)
manipulator [143](#)
[map](#) [164](#), [168](#)
 and [unordered_map](#) [166](#)
[<map>](#) [123](#)
mapped type, value [164](#)
mapping to hardware [16](#)
match
 greedy [132](#), [135](#)
 lazy [132](#), [135](#)
mathematical
 constant [234](#)
 function [228](#)
 function, standard [228](#)
 functions, special [228](#)
[<math.h>](#) [228](#)

Max Munch rule [132](#)
[\[\[maybe_unused\]\]](#) attribute [264](#)
McIlroy, Doug [259](#)
meaning, C++ [257](#)

member
 access . [23](#)
 access -> [23](#)
 class [23](#)
 data [23](#)
 destructor for [65](#)
 function [23](#)
 function, [const](#) [56](#)
 initialization, in-class [264](#)
 initializer, default [74](#)
memberwise copy [72](#)
[mem_fn\(\)](#) [217](#)
memory [79](#)
 address [16](#)
 dynamic [57](#)
 resource, polymorphic [167](#)
 safety [196](#)
[<memory>](#) [123](#), [197](#), [199](#)
[merge\(\)](#) [182](#)
[mergeable](#), concept [192](#)
[midpoint\(\)](#) [229](#)
minus operator - [7](#)
model
 C++ [262](#)
 template compilation [117](#)
modern C++ [262](#)
modularity [29](#)
module
 C++20 [33](#)
 export [33](#)
 import [33](#)
 standard library [123](#)
 std [34](#), [277](#)
 std.compat [277](#)
modulus operator % [7](#)
month [214](#)
[mopyable](#), concept [191](#)
[movable](#), concept [191](#)

- move [72, 77](#)
 - assignment [72, 78](#)
 - constructor [72, 77](#)
- move() [78, 182, 223](#)
- move_constructible, concept [191](#)
- moved-from
 - object [78](#)
 - state [224](#)
- move-only type [223](#)
- multi-line pattern [131](#)
- multimap [168](#)
- multiple
 - character sets [128](#)
 - inheritance [259](#)
 - return values [41](#)
- multiply operator * [7](#)
- multiset [168](#)
- mutex [241](#)
- <mutex> [241](#)

N

- \n, newline [3](#)
- naked
 - delete [58](#)
 - new [58](#)
- name, function [5](#)
- namespace scope [9](#)
- namespace [35](#)
 - chrono [214](#)
 - inline [264](#)
 - pmr [167](#)
 - std [3, 36, 121](#)
 - views [188](#)
- narrowing
 - = initializer [8](#)
 - conversion [8](#)
- navigation, hierarchy [67](#)
- new

- container allocator [167](#)
- naked [58](#)
- operator [57](#)
- newline \n [3](#)
- Nicholas [126](#)
- [[nodiscard]] attribute [98, 264](#)
- noexcept [50](#)
- noexcept() [263](#)
- nonhomogeneous operation [108](#)
- non-memory resource [79](#)
- non-standard library [119](#)
- Norah [208](#)
- [[noreturn]] attribute [263](#)
- normal_distribution [231](#)
- notation
 - regular expression [131](#)
 - template [105](#)
- not-equal operator != [7](#)
- notify_one(), condition_variable [245](#)
- [[no_unique_address]] attribute [265](#)
- now() [214](#)
- NULL 0, nullptr [14](#)
- nullptr [13](#)
 - function and [217](#)
 - NULL 0 [14](#)
- number
 - of C++ users [262](#)
 - random [231](#)
- Number [108](#)
 - example [108](#)
- <numbers> [234](#)
- <numeric> [229](#)
- numerical algorithm [229](#)
- numeric_limits [234](#)

O

`:o, format()` [145](#)

object [6](#)

 function [94](#)

 in container [160](#)

 moved-from [78](#)

object-oriented programming [63](#), [258](#)

`ofstream` [147](#)

open a file [151](#)

operation

 container [83](#)

 conventional [81](#)

 default [72](#)

 essential [72](#)

 file system [153](#)

 nonhomogeneous [108](#)

 path [152](#)

operator

`->` [220](#)

`+=` [7](#)

`%=` [7](#)

`*` [220](#)

`?:` [82](#)

`&`, address-of [11](#)

`()`, call [94](#)

`*`, contents-of [11](#)

`--`, decrement [7](#)

`/`, divide [7](#)

`==`, equal [7](#)

`>`, greater-than [7](#)

`>=`, greater-than-or-equal [7](#)

`++`, increment [7](#)

`>>`, input [84](#)

`<`, less-than [7](#)

`<=`, less-than-or-equal [7](#)

`"`, literal [85](#)

`-`, minus [7](#)

`%`, modulus [7](#)

`*`, multiply [7](#)

`!=`, not-equal [7](#)

`<<`, output [3](#), [84](#)

`+`, plus [7](#)

`%`, remainder [7](#)

`*=`, scaling [7](#)

`/=`, scaling [7](#)

`<=>`, spaceship [81](#)

`[]`, subscript [25](#)

 arithmetic [7](#)

 comparison [7](#), [81](#)

 declarator [12](#)

`delete[]` [57](#)

`delete` [57](#)

`new` [57](#)

 overloaded [57](#)

 overloading [80](#)

 relational [81](#)

 user-defined [57](#)

optimization, short-string [127](#)

optional [210](#)

order

`format()` argument [145](#)

 of construction [67](#)

 of destruction [67](#)

 of evaluation [8](#)

 of, public private [23](#)

`ostream` [138](#)

`<<`, output [138](#)

`<ostream>` [138](#)

`ostream_iterator` [179](#)

`ostreamstring` [147](#)

`out_of_range` [161](#)

output [138](#)

 bits of `int` [204](#)

`cout` [3](#)

`format` [143](#)–[144](#)

 of user-defined type [141](#)

 operator `<<` [3](#), [84](#)

- [ostream << 138](#)
 - [string 140](#)
- [output_iterator, concept 192](#)
- [output_range, concept 193](#)
- [overloaded operator 57](#)
- [overloaded\(\) 210](#)
- [overloading
 - \[concept based 106\]\(#\)
 - \[function 5\]\(#\)
 - \[operator 80\]\(#\)](#)
- [override 61](#)
- [overflow, buffer 200](#)
- [overview, container 168](#)
- [ownership 197](#)
- [owning 196](#)

P

- [packaged_task thread 247](#)
- [par 183](#)
- [parallel algorithm 183](#)
- [parameterized type 88](#)
- [partial_sum\(\) 229](#)
- [par_unseq 183](#)
- [passing data to task 239](#)
- [path 151
 - \[operation 152\]\(#\)](#)
- [pattern 130
 - \[\\(?: 134\]\(#\)
 - \[multi-line 131\]\(#\)](#)
- [perfect forwarding 224](#)
- [permutable, concept 192](#)
- [phone_book example 158](#)
- [pi 234](#)
- [pipeline | 188](#)
- [plus operator + 7](#)
- [pmr, namespace 167](#)
- [pointer 17
 - \[& 11\]\(#\)
 - \[* 11\]\(#\)
 - \[dangling 196\]\(#\)
 - \[smart 84, 197, 220\]\(#\)](#)
- [policy, execution 183](#)
- [polymorphic
 - \[memory resource 167\]\(#\)
 - \[type 60\]\(#\)](#)
- [pow\(\) 228](#)
- [precision, format\(\) 145](#)
- [precison\(\) 143](#)
- [precondition 45](#)
- [predicate 94, 181
 - \[type 218\]\(#\)](#)
- [predicate, concept 191](#)
- [print, regex 133](#)
- [printf\(\) 149](#)
- [private order of, public 23](#)
- [problems, header 32](#)
- [procedural programming 2](#)
- [producer\(\) consumer\(\) example 244](#)
- [program 2
 - \[exit 225\]\(#\)](#)
- [programming
 - \[generic 103, 112, 258\]\(#\)
 - \[object-oriented 63, 258\]\(#\)
 - \[procedural 2\]\(#\)](#)
- [promise
 - \[future and 245\]\(#\)
 - \[member set_exception\\(\\) 245\]\(#\)
 - \[member set_value\\(\\) 245\]\(#\)](#)
- [promise_type, coroutine 253](#)
- [pronunciation, C++ 257](#)
- [ptrdiff_t 234](#)
- [public private order of 23](#)
- [punct, regex 133](#)
- [pure virtual 60](#)

purpose, template [103](#)
push_back() [58](#), [163](#), [169](#)
 implementation [159](#)
push_front() [163](#)

Q

quick_exit() termination [225](#)

R

R" [130](#)
race, data [239](#)
RAII [58](#), [98](#), [259](#)
 and resource management [45](#)
 and try-block [48](#)
 and try-statement [45](#)
 resource acquisition [197](#)
 scoped_lock and [241](#)–[242](#)
Rand_int example [231](#)
random
 distribution [231](#)
 engine [231](#)
 number [231](#)
<random> [123](#), [231](#)
random_access_iterator, concept [192](#)
random_access_range, concept [193](#)
random_device [233](#)
random_engine seed()
range
 checking, cost of [162](#)
 checking Vec [161](#)
 concepts [193](#)
 error [200](#), [228](#)
 for statement [12](#)
range
 adaptor [187](#)
 concept [193](#)
 concept [185](#)

infinite [185](#)
range-checking, span [200](#)
range-for, span and [200](#)
<ranges> [123](#), [185](#)
 concept in [190](#)
range_value_t [109](#)
raw string literal [130](#)
reader-writer lock [242](#)
recovery, error [47](#)
recursive_directory_iterator [152](#)
reduce() [229](#)
reference [18](#)
 &&, rvalue [77](#)
 rvalue [78](#)
 to & [12](#)
regex
 * [131](#)
 [[131](#)
 + [131](#)
 . [131](#)
 ? [131](#)
 ^ [131](#)
] [131](#)
) [131](#)
 ([131](#)
 \$ [131](#)
 { [131](#)

 } [131](#)
 | [131](#)
 alnum [133](#)
 alpha [133](#)
 blank [133](#)
 cntrl [133](#)
 \D [133](#)
 \d [133](#)
 d [133](#)
 digit [133](#)

- graph [133](#)
- `\L` [133](#)
- `\l` [133](#)
- lower [133](#)
- print [133](#)
- punct [133](#)
- regular expression [130](#)
- repetition [132](#)
- `s` [133](#)
- `\s` [133](#)
- `\S` [133](#)
- space [133](#)
- `\u` [133](#)
- `\U` [133](#)
- upper [133](#)
- `\W` [133](#)
- `\w` [133](#)
- `w` [133](#)
- `xdigit` [133](#)
- `<regex>` [123](#), [130](#)
 - regular expression [130](#)
- `regex_iterator` [135](#)
- `regex_search` [130](#)
- register, removed [267](#)
- regular
 - expression notation [131](#)
 - expression `regex` [130](#)
 - expression `<regex>` [130](#)
- regular, concept [191](#)
- `regular_invocable`, concept [191](#)
- `reinterpret_cast` [59](#)
- relation, concept [191](#)
- relational operator [81](#)
- remainder operator `%` [7](#)
- `remove_const_t` [221](#)
- removed
 - assignment-to-string-literal [267](#)
 - `auto_ptr` [267](#)

- exception specification [267](#)
- export [267](#)
- feature [267](#)
- register [267](#)
- repetition, `regex` [132](#)
- `replace()` [182](#)
 - string [126](#)
- `replace_if()` [182](#)
- `request_stop()` [249](#)
- requirement, template [104](#)
- requirements [105](#)
 - element [160](#)
- requires
 - clause [105](#)
 - expression [106](#)
- `reserve()` [159](#), [169](#)
- `resize()` [169](#)
- resource
 - acquisition RAII [197](#)
 - handle [75](#), [198](#)
 - leak [67](#), [78](#), [197](#)
 - management [78](#), [197](#)
 - management, RAII and [45](#)
 - non-memory [79](#)
 - retention [79](#)
 - safe [262](#)
 - safety [78](#)
- rethrow [46](#)
- return
 - container [176](#)
 - function value [72](#)
 - type `->` [40](#)
 - type `auto` [40](#)
 - type deduction [40](#)
 - type, suffix [263](#)
 - type suffix [40](#)
 - type, `void` [4](#)
 - value, function [37](#)

- values, multiple [41](#)
- returning results from task [240](#)
- `reverse_view` [187](#)
- `rieman_zeta()` [228](#)
- Ritchie, Dennis [259](#)
- rule
 - Max Munch [132](#)
 - of zero [73](#)
- run-time
 - check [48](#)
 - error [44](#)
- rvalue
 - reference [78](#)
 - reference && [77](#)

S

- `s` literal suffix [127](#)
- `\S`, regex [133](#)
- `\s`, regex [133](#)
- `s`, regex [133](#)
- safe
 - resource [262](#)
 - type [262](#)
- safety
 - memory [196](#)
- resource [78](#)
- `same_as`, concept [190](#)
- saving space [27](#)
- scaling
 - operator `/=` [7](#)
 - operator `*=` [7](#)
- `scanf()` [149](#)
- scientific [143](#)
- scope
 - and lifetime [9](#)
 - class [9](#)
 - local [9](#)
 - namespace [9](#)
- `scoped_lock` [197](#)
 - and RAII [241–242](#)
 - `unique_lock` and [245](#)
- `scoped_lock()` [242](#)
- `scope_exit` [98](#)
- search, binary [182](#)
- `seed()`, `random_engine`
- semiregular, concept [191](#)
- Sentinel example [193](#)
- `sentinel_for`, concept [192](#)
- separate compilation [30](#)
- sequence [174](#)
 - half-open [182](#)
- Sequence example [109](#)
- `set` [168](#)
- `<set>` [123](#)
- `set_exception()`, `promise` member [245](#)
- `setstate()`, `iostream` [141](#)
- `set_value()`, `promise` member [245](#)
- SFINAE [221](#)
- `shared_lock` [242](#)
- `shared_mutex` [242](#)
- `shared_ptr` [197](#)
- sharing data task [241](#)
- short-string optimization [127](#)
- sightseeing tour
- `signed_integral`, concept [190](#)
- SIMD [183](#)
- Simula [251](#), [255](#)
- `sin()` [228](#)
- singly-linked list, `forward_list` [164](#)
- `sinh()` [228](#)
- size of type [6](#)
- `size()` [83](#), [169](#)
 - array [203](#)
- `sized_range`, concept [193](#)

- sized_sentinel_for, concept [192](#)
- sizeof [6](#)
- sizeof() [217](#)
- size_t [100](#), [234](#)
- smart pointer [84](#), [197](#), [220](#)
- smatch [130](#)
- sort() [173](#), [182](#)
- sortable, concept [192](#)
- source_location
 - file_name() [222](#)
 - function_name() [222](#)
 - line() [222](#)
- space, saving [27](#)
- space, regex [133](#)
- spaceship operator <=> [81](#)
- span
 - and range-for [200](#)
 - range-checking [200](#)
 - string_view and [200](#)
- spanstream C++23 [149](#)
- special mathematical functions [228](#)
- specialization [89](#)
- specialized container [201](#)
- sph_bessel() [228](#)
- split_view [187](#)
- sqrt() [228](#)
- <sstream> [123](#), [147](#)
- standard
 - ISO C++ [2](#)
 - library [119](#)
 - library algorithm [181](#)
 - library, C++ [2](#)
 - library, C with Classes [259](#)
 - library, C++98 [259](#)
 - library container [168](#)
 - library facilities [120](#)
 - library header [121](#), [123](#)
 - library module [123](#)

- library std [121](#)
- library suffix [121](#)
- mathematical function [228](#)
- standardization, C++ [260](#)
- state
 - I/O [141](#)
 - moved-from [224](#)
- statement
 - for [12](#)
 - if [14](#)
 - range for [12](#)
 - switch [15](#)
 - while [14](#)
- static_assert [234](#)
 - and concept [108](#)
 - assertion [50](#)
- static_cast [59](#)
- std
 - module [34](#), [277](#)
 - namespace [3](#), [36](#), [121](#)
 - standard library [121](#)
 - sub-namespaces [121](#)
- std::byte byte [224](#)
- std.compat, module [277](#)
- <stdexcept> [123](#)
- std.h [278](#)
- stem() [152](#)
- STL [259](#)
- stopping thread [248](#)
- stop_requested() [249](#)
- stop_source [249](#)
- stop_token [248](#)
- store, free [57](#)
- strict_weak_order, concept [191](#)
- string
 - C-style [13](#)
 - literal " [3](#)

- literal, raw [130](#)
- literal template argument [91](#)
- literal, type of [127](#)
- Unicode [128](#)
- string [125](#)
 - [\[\]](#) [126](#)
 - [==](#) [126](#)
 - [append +=](#) [126](#)
 - [assignment =](#) [126](#)
 - [concatenation +](#) [125](#)
 - [implementation](#) [127](#)
 - [input](#) [140](#)
 - [output](#) [140](#)
 - [replace\(\)](#) [126](#)
 - [substr\(\)](#) [126](#)
- [<string>](#) [123](#), [125](#)
- [string_literals](#), literals [127](#)
- [stringstream](#) [147](#)
- [string_view](#) [128](#)
 - [and span](#) [200](#)
- [string_view_literals](#), literals [129](#)
- [strstream](#) deprecated [148](#), [267](#)
- [struct](#) [22](#)
 - [class and](#) [25](#)
 - [union and](#) [27](#)
- [structured binding](#) [41](#)
- style
 - C++ [269](#)
 - C [269](#)
- [subclass](#), [superclass](#) and [61](#)
- [sub-namespaces](#), [std](#) [121](#)
- [subscript operator \[\]](#) [25](#)
- [subscripting](#), [\[\]](#) [169](#)
- [substr\(\)](#), [string](#) [126](#)
- [suffix](#) [84](#)
 - [return type](#) [263](#)
 - [return type](#) [40](#)
 - [s literal](#) [127](#)

- [standard library](#) [121](#)
 - [sv literal](#) [129](#)
 - [time](#) [214](#)
- [sum\(\)](#) example [104](#)
- [superclass](#) and [subclass](#) [61](#)
- [sv literal suffix](#) [129](#)
- [swap\(\)](#) [84](#)
- [swappable](#), concept [190](#)
- [swappable_with](#), concept [190](#)
- [switch statement](#) [15](#)
- [synchronized_pool_resource](#) [167](#)
- [syncstream](#) [149](#)
- [sync_with_stdio\(\)](#) [149](#)
- [syntax](#), [\[\[\]\]](#) attribute [263](#)
- [system_clock](#) [214](#)

T

- [table](#), [hash](#) [165](#)
- [tagged union](#) [28](#)
- [take\(\)](#) [189](#)
- [take_view](#) [186–187](#)
- [tanh\(\)](#) [228](#)
- [task](#)
 - [and thread](#) [238](#)
 - [communication](#) [245](#)
 - [passing data to](#) [239](#)
 - [returning results from](#) [240](#)
 - [sharing data](#) [241](#)
- [task example](#) [253](#)
- [tau example](#) [235](#)
- [TC++PL](#) [256](#)
- [template](#)
 - [argument](#), [string literal](#) [91](#)
 - [type safty](#) [90](#)
- [template](#) [87](#)
 - [..., variadic](#) [114](#)
 - [alias](#) [100](#)
 - [argument](#), [constrained](#) [90](#)

- argument, default [108](#)
- class [88](#)
- compilation model [117](#)
- constrained [90](#)
- debugging [113](#)
- definition checking [109](#)
- extern [264](#)
- function [93](#)
- instantiation time [117](#)
- notation [105](#)
- purpose [103](#)
- requirement [104](#)
- variable [99](#)
- virtual [94](#)
- terminate() termination [225](#)
- termination [48](#)
 - exit() [225](#)
 - quick_exit() [225](#)
 - terminate() [225](#)
- this [76](#)
- [this] and [*this] [95](#)
- thread
 - join() [238](#)
 - packaged_task [247](#)
 - stopping [248](#)
 - task and [238](#)
- <thread> [123](#), [238](#)
- thread_local [264](#)
- three_way_comparable, concept [190](#)
- three_way_comparable_with, concept [190](#)
- time [214](#)
 - suffix [214](#)
 - template instantiation [117](#)
- timeline, C++ [256](#)
- time_point [214](#)
- time_zone [216](#)
- timing, clock [243](#)

- to hardware, mapping [16](#)
- totally_ordered, concept [190](#)
- totally_ordered_with, concept [190](#)
- tour, sightseeing
- transform_reduce() [229](#)
- transform_view [187](#)
- translation unit [32](#)
- try
 - block [44](#)
 - block as function body [161](#)
- try-block, RAII and [48](#)
- try-statement, RAII and [45](#)
- type [6](#)
 - alias [234](#)
 - argument [90](#)
 - associate [222](#)
 - built-in [21](#)
 - concept and [111](#)
 - concepts [190](#)
 - concrete [54](#)
 - conversion, explicit [59](#)
 - file [154](#)
 - function [217](#)
 - fundamental [6](#)
 - generator [221](#)
 - get<>() by [207](#)
 - input of user-defined [141](#)
 - move-only [223](#)
 - of string literal [127](#)
 - output of user-defined [141](#)
 - parameterized [88](#)
 - polymorphic [60](#)
 - predicate [218](#)
 - safe [262](#)
 - safety template [90](#)
 - size of [6](#)
 - user-defined [21](#)
- typename [88](#), [177](#)

[<type_traits> 218](#)
typing, duck [117](#)

U

[\U, regex 133](#)
[\u, regex 133](#)
UDL, user-defined literal [84](#)
[uint_least64_t 234](#)
[unget\(\), istream 141](#)
Unicode string [128](#)
[uniform_int_distribution 231](#)
uninitialized [8](#)
union [27](#)
 and struct [27](#)
 and variant [28](#)
 tagged [28](#)
[unique_copy\(\) 173, 182](#)
[unique_lock 242, 244](#)
 and [scoped_lock 245](#)
[unique_ptr 68, 197](#)
[\[\[unlikely\]\] attribute 265](#)
[unordered_map 165, 168](#)
 [hash<> 84](#)
 map and [166](#)
[<unordered_map> 123](#)
[unordered_multimap 168](#)
[unordered_multiset 168](#)
[unordered_set 168](#)
[unsigned 6](#)
[unsigned_integral, concept 190](#)
[upper, regex 133](#)
use
 C++ [262](#)
 concept [104](#)
user-defined
 literal [264](#)
 literal UDL [84](#)
 operator [57](#)

type [21](#)
type, input of [141](#)
type, output of [141](#)

using
 alias [100](#)
 declaration [36](#)
 directive [36](#)
 enum [26](#)
 usual arithmetic conversions [7](#)
[<utility> 123, 206](#)

V

[valarray 233](#)
[<valarray> 233](#)
value [6](#)
 argument [90](#)
 key and [164](#)
 mapped type [164](#)
 return, function [72](#)

values, multiple return [41](#)
[Value_type example 109](#)
[value_type 100, 169](#)
 variable [5–6](#)
 concept and [111](#)
 template [99](#)
variadic template ... [114](#)
variant [209](#)
 union and [28](#)
Vec
 example [161](#)
 range checking [161](#)
vector arithmetic [233](#)
[Vector example 22–23, 29, 33–34, 57–58, 73, 75, 77, 88–89, 91–92, 100](#)
vector [158, 168](#)
 array vs. [203](#)

`<vector>` [123](#)
`vector<bool>` [201](#)
vectorized [183](#)
`vformat()` [146](#)
view [186](#)
view, concept [193](#)
views, namespace [188](#)
virtual [60](#)
 destructor [65](#)
 function, implementation of [62](#)
 function table `vtbl` [62](#)
 pure [60](#)
 template [94](#)
`void*` incompatibility [270](#)
`void` return type [4](#)
`vtbl`, virtual function table [62](#)

zero, rule of [73](#)
`zoned_time` [216](#)

W

`w`, regex [133](#)
`\W`, regex [133](#)
`\w`, regex [133](#)
`wait()`, `condition_variable` [244](#)
weekday [214](#)
WG21 [256](#)
`while` statement [14](#)
whitespace [139](#)

X

`:x`, `format()` [145](#)
X3J16 [260](#)
`xdigit`, regex [133](#)

Y

year [214](#)

Z