C++ для профи

Молниеносный старт

Джош Лоспинозо

C++ CRASH COURSE

A Fast-Paced Introduction

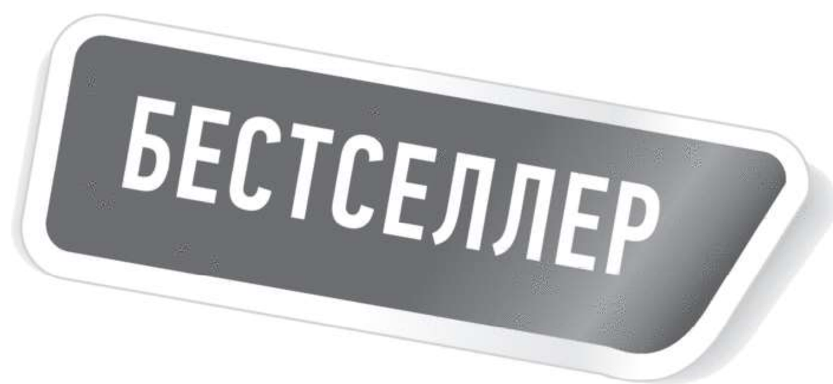
by Josh Lospinoso

no starch press
San Francisco

ДЖОШ ЛОСПИНОЗО

C++ ДЛЯ ПРОФИ

МОЛНИЕНОСНЫЙ СТАРТ



Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2021

ББК 32.973.2-018.1
УДК 004.43
Л79

Лоспинозо Джош

Л79 C++ для профи. — СПб.: Питер, 2021. — 816 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1730-7

C++ — популярный язык для создания ПО. В руках увлеченного программиста C++ становится прекрасным инструментом для создания лаконичного, эффективного и читаемого кода, которым можно гордиться. «C++ для профи» адресован программистам среднего и продвинутого уровней, вы продеретесь сквозь тернии к самому ядру C++. Часть 1 охватывает основы языка C++ — от типов и функций до жизненного цикла объектов и выражений. В части 2 представлена стандартная библиотека C++ и библиотеки Boost. Вы узнаете о специальных вспомогательных классах, структурах данных и алгоритмах, а также о том, как управлять файловыми системами и создавать высокопроизводительные программы, которые обмениваются данными по сети.

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

ББК 32.973.2-018.1
УДК 004.43

Права на издание получены по соглашению с No Starch Press. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1593278885 англ.
ISBN 978-5-4461-1730-7

© 2019 by Josh Lospinoso. C++ Crash Course: A fast-paced introduction. ISBN 978-1-59327-885-5, published by No Starch Press.
© Перевод на русский язык ООО Издательство «Питер», 2021
© Издание на русском языке, оформление ООО Издательство «Питер», 2021
© Серия «Для профессионалов», 2021

Краткое содержание

Часть 1. Основы языка C++ 57

Глава 1. Создаем и запускаем 59

Глава 2. Типы 87

Глава 3. Ссылочные типы 126

Глава 4. Жизненный цикл объекта 148

Глава 5. Полиморфизм во время выполнения 195

Глава 6. Полиморфизм во время компиляции 211

Глава 7. Выражения 246

Глава 8. Инструкции 278

Глава 9. Функции 311

Часть 2. Библиотеки и фреймворки C++ 349

Глава 10. Тестирование 351

Глава 11. Умные указатели 413

Глава 12. Утилиты 444

Глава 13. Контейнеры 486

Глава 14. Итераторы 547

Глава 15. Строки 566

Глава 16. Потоки 612

Глава 17. Файловые системы 642

Глава 18. Алгоритмы 665

Глава 19. Конкурентность и параллелизм 736

Глава 20. Сетевое программирование с помощью Boost Asio 762

Глава 21. Создание приложений 791

Оглавление

Об авторе **24**

О научном редакторе **24**

Предисловие **25**

Благодарности **29**

От издательства **30**

Введение **31**

Об этой книге 32

Кому будет интересна эта книга? 32

Структура книги 33

Часть I. Основы языка C++ 33

Часть II. Библиотеки и фреймворки C++ 34

Увертюра для C-программистов **36**

Обновление до Super C 38

Перегрузка функций 38

Ссылки 39

Инициализация с помощью auto 41

Пространства имен и неявный typedef для struct, union и enum 42

Смешение объектных файлов C и C++ 44

Темы C++ 46

Краткое изложение идей и повторное использование кода 46

Стандартная библиотека C++ 48

Лямбда-выражения 49

Обобщенное программирование с помощью шаблонов 50

Инварианты классов и управление ресурсами 51

Семантика перемещения 55

Расслабьтесь и получайте удовольствие 56

Часть 1. Основы языка C++ 57

Глава 1. Создаем и запускаем 59

Структура базовой программы на C++ 60

Создание первого исходного файла на C++ 60

Метод main: точка запуска программы 60

Библиотеки: добавление внешнего кода 61

Цепочка инструментов компилятора 61

Настройка среды разработки 62

Windows 10 и выше: Visual Studio 62

macOS: Xcode 64

Linux и GCC 65

Установка GCC и Clang в Debian 66

Установка GCC из источника 67

Текстовые редакторы 69

Инициализация C++ 70

Система типов C++ 70

Объявление переменных 70

Инициализация состояния переменной 70

Условные выражения 71

Функции 73

Спецификаторы формата printf 75

Пересмотр step_function 75

Комментарии 77

Отладка 78

Visual Studio 78

Xcode 80

Отладка GCC и Clang с помощью GDB и LLDB 82

Итоги 85

Упражнения 85

Что еще почитать? 86

Глава 2. Типы 87

Основные типы 87

Целочисленные типы 88

Типы с плавающей точкой 91

Символьные типы ... 93
Логические типы ... 95
Тип std::byte ... 97
Тип size_t ... 98
void ... 99
Массивы ... 100
Инициализация массива ... 100
Доступ к элементам массива ... 100
Экскурсия по циклу for ... 101
Строки в стиле C ... 103
Пользовательские типы ... 107
Типы перечислений ... 107
Простые классы ... 110
Объединения ... 112
Полнофункциональные классы в C++ ... 113
Методы ... 113
Контроль доступа ... 114
Конструкторы ... 116
Инициализация ... 118
Деструктор ... 124
Итоги ... 124
Упражнения ... 125
Что еще почитать? ... 125
Глава 3. Ссылочные типы ... 126
Указатели ... 126
Обращение к переменным ... 127
Разыменование указателей ... 128
Оператор «стрелка» ... 130
Указатели и массивы ... 131
Опасность указателей ... 133
Указатели void и std::byte ... 135
nullptr и логические выражения ... 135
Ссылки ... 136
Использование указателей и ссылок ... 137
Связные списки: каноническая структура данных на основе указателей ... 137
Использование ссылок ... 139

Указатели this ... 140
Правильное использование const ... 141
Переменные-члены const ... 142
Списки инициализаторов членов ... 143
Вывод типов с помощью auto ... 144
Инициализация с помощью auto ... 144
auto и ссылочные типы ... 145
auto и рефакторинг кода ... 145
Итоги ... 146
Упражнения ... 146
Что еще почитать? ... 147
Глава 4. Жизненный цикл объекта ... 148
Длительность хранения объекта ... 148
Выделение, освобождение и срок службы ... 148
Управление памятью ... 149
Автоматическая длительность хранения ... 149
Статическая длительность хранения ... 150
Локальная потоковая длительность хранения ... 153
Динамическая длительность хранения ... 154
Отслеживание жизненного цикла объекта ... 156
Исключения ... 158
Ключевое слово throw ... 159
Использование блоков try-catch ... 159
Классы исключений stdlib ... 160
Обработка исключений ... 163
Пользовательские исключения ... 164
Ключевое слово noexcept ... 165
Исключения и стеки вызовов ... 165
Класс SimpleString ... 168
Добавление и вывод ... 169
Использование SimpleString ... 170
Составление SimpleString ... 171
Размотка стека вызовов ... 172
Исключения и производительность ... 174
Альтернативы для исключений ... 175

Семантика копирования ... 176
Конструкторы копирования ... 178
Присваивание копии ... 181
Копирование по умолчанию ... 183
Руководство по копированию ... 184
Семантика перемещения ... 184
Копирование может быть расточительным ... 185
Категории значений ... 186
Ссылки на l-значения и r-значения ... 187
Функция std::move ... 188
Конструктор переноса ... 188
Присваивание перемещения ... 189
Конечный продукт ... 191
Методы, генерируемые компилятором ... 192
Итоги ... 193
Упражнения ... 194
Что еще почитать? ... 194
Глава 5. Полиморфизм во время выполнения ... 195
Полиморфизм ... 195
Пример для мотивации ... 196
Добавление новых регистраторов ... 198
Интерфейсы ... 199
Композиция объектов и реализация наследования ... 199
Определение интерфейсов ... 200
Базовое наследование классов ... 200
Наследование членов ... 201
Методы virtual ... 202
Чисто виртуальные классы и виртуальные деструкторы ... 204
Реализация интерфейсов ... 206
Использование интерфейсов ... 206
Обновление банкового регистратора ... 207
Внедрение через конструктор ... 207
Внедрение через свойство ... 208
Выбор между внедрением через конструктор или свойство ... 209
Итоги ... 210
Упражнения ... 210
Что еще почитать? ... 210

Глава 6. Полиморфизм во время компиляции 211
Шаблоны 211
Объявление шаблонов 212
Определения класса шаблона 212
Определения функции шаблона 212
Создание экземпляров шаблонов 213
Именованные функции преобразования 213
const_cast 214
static_cast 214
reinterpret_cast 215
narrow_cast 216
mean: пример функции шаблона 218
Обобщение mean 218
Вывод типа шаблона 221
SimpleUniquePointer: пример класса шаблона 222
Проверка типов в шаблонах 224
Концепты 226
Определение концепта 227
Типажи типа 227
Требования 229
Создание концепта из выражений требования 231
Использование концептов 232
Специальные выражения требований 236
static_assert: полумеры до концептов 237
Нетиповые параметры шаблона 238
Вариативные шаблоны 241
Продвинутое использование шаблонов 242
Специализация шаблона 242
Связка имен 242
Функция типа 242
Метапрограммирование шаблонов 243
Организация исходного кода шаблона 243
Полиморфизм во время выполнения и компиляции 243
Итоги 244
Упражнения 244
Что еще почитать? 245

Глава 7. Выражения ... 246
Операторы ... 246
Логические операторы ... 247
Арифметические операторы ... 247
Операторы присваивания ... 249
Операторы увеличения и уменьшения ... 250
Операторы сравнения ... 250
Операторы доступа к членам ... 250
Тернарные условные операторы ... 251
Оператор запятой ... 252
Перегрузка операторов ... 253
Оператор перегрузки new ... 254
Приоритет операторов и ассоциативность ... 260
Порядок вычисления ... 263
Пользовательские литералы ... 264
Преобразования типов ... 265
Неявные преобразования типов ... 265
Явное преобразование типов ... 268
Приведения в стиле C ... 269
Пользовательские приведения типов ... 270
Постоянные выражения ... 271
Красочный пример ... 272
Использование constexpr ... 275
Изменчивые выражения ... 275
Итоги ... 276
Упражнения ... 276
Что еще почитать? ... 277

Глава 8. Инструкции ... 278
Инструкции-выражения ... 278
Составные операторы ... 279
Операторы объявлений ... 280
Функции ... 280
Пространства имен ... 283
Совмещение имен типов ... 287
Структурированные привязки ... 289
Атрибуты ... 291

Операторы выбора 292
Оператор if........ 292
Операторы switch 296
Операторы перебора 298
Циклы while........ 298
Циклы do-while........ 299
Циклы for 300
Циклы for на основе диапазонов 302
Инструкции перехода 306
Операторы break 306
Инструкции continue........ 307
Инструкции goto........ 308
Итоги........ 309
Упражнения 309
Что еще почитать? 310
Глава 9. Функции 311
Объявления функций 311
Префиксные модификаторы 312
Суффиксные модификаторы........ 313
Возвращаемые типы auto 315
auto и шаблоны функций........ 316
Разрешение перегрузки........ 317
Вариативные функции........ 318
Вариативные шаблоны 320
Программирование с помощью блоков параметров 320
Пересмотр функции sum 321
Выражения свертки........ 322
Указатели функций........ 322
Объявление указателя функции 323
Совмещение имен типов и указатели функций 324
Оператор вызова функции........ 324
Пример подсчета........ 325
Лямбда-выражения 327
Использование........ 327
Тела и параметры лямбда-выражений........ 328

Параметры по умолчанию 329
Обобщенные лямбда-выражения 330
Возвращаемые типы лямбда-выражений 331
Захват лямбда-выражений 332
Лямбда-выражения с constexpr 338
std::function 338
Объявление функции 339
Пустые функции 339
Расширенный пример 340
Функция main и командная строка 342
Три перегрузки main 342
Изучение параметров программ 343
Еще больше примеров 344
Статус выхода 347
Итоги 347
Упражнения 347
Что еще почитать? 348

Часть 2. Библиотеки и фреймворки C++ 349

Глава 10. Тестирование 351
Юнит-тестирование 351
Интеграционное тестирование 352
Приемочное тестирование 352
Тестирование производительности 352
Расширенный пример: AutoBrake 353
Реализация AutoBrake 355
Разработка через тестирование 356
Добавление интерфейса Service-Bus 367
Фреймворки юнит-тестирования и имитации 374
Google Test 381
Boost Test 389
Итоги: фреймворки тестирования 394
Фреймворки имитации 395
Google Mock 396
HippoMocks 405
Несколько слов о других вариантах имитации: FakeIt и Trompeloeil 410

Итоги ... 410
Упражнения ... 411
Что еще почитать? ... 412

Глава 11. Умные указатели ... 413

Умные указатели ... 413
Владение умными указателями ... 414
Ограниченные указатели ... 414
Создание ... 415
Добавление OathBreakers ... 415
Явное приведение логических типов на основе владения ... 416
Обертка RAII ... 416
Семантика указателей ... 417
Сравнение с nullptr ... 418
Обмен ... 418
Сброс и замена scoped_ptr ... 419
Непереносимость ... 420
boost::scoped_array ... 420
Неполный список поддерживаемых операций ... 421
Уникальные указатели ... 422
Создание ... 422
Поддерживаемые операции ... 422
Переносимое и исключительное владение ... 423
Уникальные массивы ... 423
Удалители ... 424
Пользовательские удалители и системное программирование ... 425
Неполный список поддерживаемых операций ... 427
Общие указатели ... 429
Создание ... 429
Определение распределителя ... 430
Поддерживаемые операции ... 431
Переносимое и неисключительное владение ... 431
Общие массивы ... 432
Удалители ... 432
Неполный список поддерживаемых операций ... 432
Слабые указатели ... 434
Создание ... 435
Получение временного владения ... 435

Продвинутые шаблоны ... 436
Поддерживаемые операции ... 436
Навязчивые указатели ... 436
Обзор вариантов умных указателей ... 439
Распределители ... 439
Итоги ... 442
Упражнения ... 442
Что еще почитать? ... 443

Глава 12. Утилиты ... 444

Структуры данных ... 444
tribool ... 445
optional ... 447
pair ... 450
tuple ... 452
any ... 453
variant ... 454
Дата и время ... 459
DateTime в Boost ... 459
Chrono ... 463
Числовые данные ... 468
Числовые функции ... 469
Комплексные числа ... 470
Математические постоянные ... 472
Случайные числа ... 473
Числовые ограничения ... 479
Numeric Conversion в Boost ... 480
Рациональная арифметика во время компиляции ... 482
Итоги ... 484
Упражнения ... 484
Что еще почитать? ... 485

Глава 13. Контейнеры ... 486

Контейнеры последовательностей ... 487
Массивы ... 487
Векторы ... 495
Узкоспециализированные контейнеры последовательности ... 504

Ассоциативные контейнеры ... 516
Множества ... 517
Неупорядоченные множества ... 524
Ассоциативные массивы ... 529
Специализированные ассоциативные контейнеры ... 537
Графы и деревья свойств ... 538
Библиотека Boost Graph ... 539
Деревья свойств в Boost ... 540
Списки инициализаторов ... 541
Итоги ... 543
Упражнения ... 544
Что еще почитать? ... 546
Глава 14. Итераторы ... 547
Категории итераторов ... 547
Итераторы вывода ... 548
Итераторы ввода ... 550
Однонаправленные итераторы ... 552
Двунаправленные итераторы ... 553
Итераторы с произвольным доступом ... 554
Непрерывные итераторы ... 556
Изменяемые итераторы ... 556
Вспомогательные функции итераторов ... 557
std::advance ... 557
std::next и std::prev ... 559
std::distance ... 560
std::iter_swap ... 561
Дополнительные итераторные адаптеры ... 561
Итераторные адаптеры переноса ... 561
Обратные итераторные адаптеры ... 563
Итоги ... 564
Упражнения ... 565
Что еще почитать? ... 565
Глава 15. Строки ... 566
std::string ... 566
Создание ... 567

Хранение строк и оптимизация небольших строк 570
Поэлементный и итераторный доступ........ 572
Сравнение строк 573
Управление элементами........ 575
Поиск........ 580
Числовые преобразования 584
Строковое представление........ 586
Создание........ 587
Поддерживаемые операции string_view 588
Владение, использование и эффективность 589
Регулярные выражения 590
Шаблоны........ 590
basic_regex........ 592
Алгоритмы 593
Библиотека Boost String Algorithms 597
Boost Range........ 598
Предикаты 598
Классификаторы 600
Искатели 601
Алгоритмы изменения 603
Разделение и соединение........ 606
Поиск 607
Boost Tokenizer 609
Локализация 610
Итоги........ 610
Упражнения 610
Что еще почитать? 611
Глава 16. Потоки 612
Потоки 612
Классы потоков 613
Состояние потока........ 619
Буферизация и сброс 622
Манипуляторы........ 622
Пользовательские типы........ 625
Строковые потоки 628
Файловые потоки 631

Буферы потоков 636
Произвольный доступ 638
Итоги 640
Упражнения 641
Что еще почитать? 641

Глава 17. Файловые системы 642
Концепты файловых систем 642
std::filesystem::path 643
Создание path 644
Декомпозиция path 644
Изменение path 645
Обзор методов path файловой системы 647
Файлы и каталоги 648
Обработка ошибок 649
Функции композиции path 649
Просмотр типов файлов 650
Просмотр файлов и каталогов 652
Управление файлами и каталогами 653
Итераторы каталогов 656
Создание 656
Записи каталогов 657
Рекурсивный перебор каталога 659
Взаимосовместимость в fstream 662
Итоги 663
Упражнения 663
Что еще почитать? 664

Глава 18. Алгоритмы 665
Сложность алгоритмов 666
Политика выполнения 667
Операции, не изменяющие последовательность 668
all_of 668
any_of 669
none_of 670
for_each 671
for_each_n 672

find, find_if и find_if_not ... 673
find_end ... 675
find_first_of ... 676
adjacent_find ... 677
count ... 678
mismatch ... 679
equal ... 680
is_permutation ... 681
search ... 682
search_n ... 683
Операции, изменяющие последовательность ... 684
copy ... 684
copy_n ... 685
copy_backward ... 686
move ... 687
move_backward ... 689
swap_ranges ... 690
transform ... 691
replace ... 693
fill ... 694
generate ... 695
remove ... 697
unique ... 698
reverse ... 699
sample ... 700
shuffle ... 703
Сортировка и связанные операции ... 705
sort ... 705
stable_sort ... 706
partial_sort ... 708
is_sorted ... 710
nth_element ... 711
Бинарный поиск ... 712
lower_bound ... 712
upper_bound ... 713
equal_range ... 714
binary_search ... 714
Алгоритмы разбиения ... 715

is_partitioned 716
partition 717
partition_copy 718
stable_partition 719
Алгоритмы слияния 720
merge 720
Алгоритмы предельных значений 721
min и max 721
min_element и max_element 722
clamp 724
Числовые операции 724
Полезные операторы 725
iota 725
accumulate 726
reduce 727
inner_product 728
adjacent_difference 729
partial_sum 730
Другие алгоритмы 731
Boost Algorithm 733
Что еще почитать? 735
Глава 19. Конкурентность и параллелизм 736
Конкурентное программирование 736
Асинхронные задачи 737
Совместное использование и координирование 744
Низкоуровневые средства конкурентного программирования 756
Параллельные алгоритмы 757
Пример: параллельная сортировка 757
Параллельные алгоритмы — это не магия 759
Итоги 760
Упражнения 760
Что еще почитать? 761
Глава 20. Сетевое программирование с помощью Boost Asio 762
Модель программирования Boost Asio 763
Сетевое программирование с помощью Asio 765

Модуль интернет-протокола ... 765
Разрешение имени хост-системы ... 767
Соединение ... 769
Буферы ... 771
Чтение и запись данных с помощью буферов ... 774
Протокол передачи гипертекста (HTTP) ... 776
Реализация простого HTTP-клиента в Boost Asio ... 777
Асинхронные чтение и запись ... 779
Создание сервера ... 783
Конкурентный режим в Boost Asio ... 788
Итоги ... 789
Упражнения ... 790
Что еще почитать? ... 790
Глава 21. Создание приложений ... 791
Поддержка программ ... 791
Обработка завершения программы и очистка ... 793
Коммуникация с окружающей средой ... 797
Управление сигналами операционной системы ... 799
ProgramOptions в Boost ... 800
Описание опций ... 801
Разбор опций ... 804
Совместное использование опций и доступ к ним ... 805
Соединяем все вместе ... 806
Отдельные моменты компиляции ... 809
Пересмотр препроцессора ... 809
Оптимизация компилятора ... 812
Связь с C ... 813
Итоги ... 814
Упражнения ... 814
Что еще почитать? ... 815

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

int main() {
    auto i{ 0x01B99644 };
    std::string x{ " DFaeeillnor" };
    while (i--) std::next_permutation(x.begin(), x.end());
    std::cout << x;
}
```

Об авторе

Джош Лоспинозо (Josh Lospinoso) — доктор философии и предприниматель, прослуживший 15 лет в армии США. Джош — офицер, занимающийся вопросами кибербезопасности. Написал десятки программ для средств информационной безопасности и преподавал C++ начинающим разработчикам. Выступает на различных конференциях, является автором более 20 рецензируемых статей и стипендиатом Родса, а также имеет патент. В 2012 году стал соучредителем успешной охранной компании. Джош ведет блог и активно участвует в разработке ПО с открытым исходным кодом.

О научном редакторе

Кайл Уиллмон (Kyle Willmon) — разработчик информационных систем с 12-летним опытом в C++. В течение 7 лет работал в сообществе по информационной безопасности, используя C++, Python и Go в различных проектах. В настоящее время является разработчиком в команде Sony Global Threat Emulation.

Предисловие

«C++ — сложный язык». Эту репутацию C++ заработал за несколько десятилетий, причем не совсем справедливо. Часто это утверждение считают поводом не изучать C++ или причиной, по которой стоит выбрать другой язык программирования. Такие аргументы трудно обосновать, ибо основная предпосылка неверна: C++ — не сложный язык. Первая проблема C++ — это его репутация. Вторая — отсутствие качественных учебных материалов для его изучения.

Сам язык за последние четыре десятилетия эволюционировал из языка C. Он начинался как ответвление C (с небольшими дополнениями) и предкомпилятор под названием Cfront, компилирующий ранний код C++ в код C, который затем обрабатывался с помощью компилятора C. Отсюда и название Cfront — «перед C». Спустя несколько лет оказалось, что это решение слишком ограничивает язык, и была предпринята работа по созданию фактического компилятора. Компилятор, написанный Бьёрном Страуструпом (Bjarne Stroustrup) (изобретатель языка), мог компилировать программу на C++ отдельно. Другие компании также были заинтересованы в продолжении базовой поддержки C и создали свои собственные компиляторы C++, в основном совместимые с Cfront или более новым компилятором.

Такой подход оказался несостоятельным, так как язык был непереносимым и абсолютно несовместимым между компиляторами, не говоря уже о том, что хранение всех решений и указаний в руках одного человека не способствовало созданию международного стандарта между компаниями — для этого существуют стандартные процедуры и организации, которые ими управляют. Таким образом, C++ стал стандартом ISO. После нескольких лет разработки в 1998 году вышел первый официальный стандарт C++, и люди возрадовались.

Но радовались недолго: хотя C++ 98 и был хорошим определением, он включал в себя несколько новых неожиданных разработок и имел пару функций, которые между собой весьма странно взаимодействовали. В некоторых случаях сами функции были хорошо написаны, но взаимодействие между общими функциями просто отсутствовало: например, возможность иметь имя файла в виде `std :: string` и затем открывать файл с ним.

Другим запоздалым дополнением стала поддержка шаблонов — основная базовая технология, поддерживающая библиотеку стандартных шаблонов, одну из самых важных частей C++ на сегодняшний день. Только после ее выпуска люди обнаружили, что она сама по себе является полной по Тьюрингу и что многие сложные конструкции могут быть вычислены во время компиляции. Это значительно расширило возможности авторов библиотек при написании универсального кода,

который мог бы произвольно обрабатывать сложные выводы, что было не похоже на то, что могли делать другие существующие в то время языки.

Последняя сложность заключалась в том, что хотя C++ 98 и был хорош, многие компиляторы не подходили для реализации шаблонов. Два главных компилятора того времени, GNU GCC 2.7 и Microsoft Visual C++ 6.0, не смогли выполнить двухэтапный поиск имени, требуемый шаблонами. Единственным способом реализовать это право стало бы полное переписывание компилятора...

GNU пытался и дальше добавлять функции в существующую кодовую базу, но в конце концов перешел к переписыванию компилятора в версии 2.95. Долгие годы новых функций или релизов не было, и многие были недовольны этим. Некоторые компании взяли кодовую базу и попытались продолжить ее разработку, создав 2.95.2, 2.95.3 и 2.96 — все три версии известны своей нестабильностью. Наконец, вышел полностью переписанный GCC 3.0. Первоначально этот релиз был не очень успешным. Он собирал шаблоны и код C++ намного лучше, чем когда-либо делал 2.95, но не собирал ядро Linux в работающий двоичный файл. Сообщество Linux возражало против изменения своего кода для адаптации к новому компилятору, настаивая на том, что сам компилятор не работает. В конце концов, ко времени версии 3.2 сообщество сдалось и мир Linux снова сконцентрировался вокруг GCC 3.2 и выше.

Microsoft как мог старался избежать переписывания своего компилятора. Он добавлял граничные случаи к граничным случаям и эвристику, чтобы угадать, должно ли что-то быть разрешено в первом или втором проходе поиска имени шаблона. Такой подход почти сработал, но библиотеки, написанные в начале 2010-х годов, показали, что никакого возможного способа заставить их всех работать не было — даже с изменениями исходного кода. Корпорация Microsoft наконец-то переписала свой парсер и выпустила обновленную версию в 2018 году, но многие пользователи новый парсер не активировали. В 2019 году новый парсер был окончательно включен по умолчанию для новых проектов.

Ранее, в 2011 году, произошло важное событие: релиз C++ 11. После выхода C++ 98 были предложены и разработаны новые важные функции. Но из-за того, что одна функция работала не совсем так, как ожидалось, новый релиз был отложен до 2009 года. За это время были предприняты попытки заставить его работать с новой функцией. В 2009 году она была окончательно удалена, все остальное было исправлено, а версия C++ 1998 года была обновлена. Было добавлено множество новых функций и улучшений библиотеки. Компиляторы снова медленно догоняли релизы, и большинство компиляторов смогло скомпилировать большую часть C++ 11 только к концу 2013 года.

Комитет C++ извлек уроки из собственного раннего провала и представил план создания нового релиза каждые три года. План состоял в том, чтобы продумать и протестировать новые функции в течение одного года, как следует интегрировать их в следующем, стабилизировать и официально выпустить на третий год, а затем повторять этот процесс каждые три года. C++ 11 был первым экземпляром,

а 2014 год — предполагаемым годом для второго выпуска. К своей чести, комитет выполнил то, что обещал, подготовив серьезное обновление по сравнению с C++ 11 и сделав функционал C++ 11 намного более удобным, чем ранее. В большинстве случаев, где были введены осторожные ограничения, их перенесли на то, что тогда считалось приемлемым, в частности на ограничения вокруг `constexpr`.

Авторы компиляторов, которые все еще пытались заставить работать все функции C++ 11 хорошо, поняли, что нужно сменить темп, иначе они рискуют остаться позади. К 2015 году все компиляторы поддерживали практически всё в C++ 14, что можно счесть за подвиг, учитывая историю C++ 98 и C++ 11. Это также повлияло на возобновление участия в комитете по C++ всех основных авторов компиляторов — если вы знаете о функции до ее релиза, то можете стать ведущим компилятором, поддерживающим ее. Если вы обнаружите, что определенная функция не соответствует дизайну компилятора, то можете повлиять на комитет C++, настроив функцию так, чтобы ее было легче поддерживать, — таким образом вы дадите людям шанс использовать ее как можно раньше.

C++ переживает второе рождение, что началось примерно в 2011 году, когда был представлен C++ 11 и использован добавленный им стиль программирования «Modern C++». Однако он улучшился только потому, что все идеи из C++ 11 были точно настроены в C++ 14 и C++ 17, и компиляторы стали полностью поддерживать все ожидаемые функции. Более того, скоро будет выпущен новый стандарт для C++ 20 и все компиляторы в его самых современных версиях уже поддерживают его основные части.

Современный C ++ позволяет разработчикам избавиться от большинства первоначальных проблем, связанных с попытками сначала изучить C, затем C ++ 98, затем C ++ 11, а затем отучиться от всех исправленных частей C и C ++ 98. Большинство курсов начиналось с истории C++, так как было важно понять, почему что-то в нем кажется таким странным. Что касается этой книги, я включил эту информацию в предисловие, потому что Джош по праву не затронул ее.

Больше не нужно знать эту историю, чтобы начать изучать C++. Современный стиль C++ позволяет полностью пропустить ее и писать хорошо спроектированные программы, зная только основные принципы C++. Лучшее время для изучения C++ — сейчас.

Теперь вернемся к раннее упомянутому моменту — отсутствию качественных образовательных возможностей и материалов для изучения C++. В самом комитете C++ предоставляется высококачественное обучение C++ — есть учебная группа, занимающаяся исключительно преподаванием C++!

В отличие от всех других книг по C++, которые я читал, эта книга учит основам и принципам. Она учит принципам мышления, которые позволят решать задачи и учитывать возможности библиотек стандартных шаблонов. Возможно, результаты вы получитие не сразу, но когда увидите, как компилируются и запускаются первые программы, а вы полностью понимаете, как работает C++, то степень удовлетворенности будет выше. Эта книга включает в себя даже такие темы, которые

большинство книг по C++ игонирирует: настройка среды и тестирование кода перед запуском целой программы.

Наслаждайтесь чтением этой книги, делайте упражнения. А я желаю вам удачи в путешествии по C++!

Питер Биндельс (Peter Bindels),
главный инженер-программист, TomTom

Благодарности

Прежде всего благодарю свою семью за предоставленное мне творческое пространство. Написание книги заняло вдвое больше времени, чем я планировал, и я неизмеримо благодарен вам за ваше терпение.

Я в долгу перед Кайлом Уилмоном и Аароном Бреем (Aaron Bray), которые научили меня C++; Тайлеру Ортману (Tyler Ortman), который вел эту книгу от замысла до реального результата; Биллу Поллоку (Bill Pollock), который «причесал» мой стиль изложения; Крису Кливленду (Chris Cleveland), Патрику Де Хусто (Patrick De Justo), Энн Мари Уокер (Anne Marie Walker), Энни Чой (Annie Choi), Мег Снирингер (Meg Sneeringer) и Райли Хоффман (Riley Hoffman), чья первоклассная работа над текстом принесла огромную пользу этой книге, и многим первым читателям, оставившим неоценимые комментарии к сырым главам.

И наконец, я благодарю Джеффа Лоспинозо (Jeff Lospinoso), который завещал своему наивному десятилетнему племяннику зачитанную, испачканную кофе верблюжью книгу[1], с которой все и началось.

[1] *Уолл Л., Кристиансен Т., Орвант Д.* Программирование на Perl / Пер. с англ. — СПб.: Символ Плюс, 2002. — 1152 с.

От издательства

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

Мы будем рады узнать ваше мнение!

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

Введение

Хватай кисть и рисуй вместе с нами.

Боб Росс

Спрос на программирование систем огромен. Возможно, никогда еще не было лучшего времени, чтобы стать системным программистом, учитывая распространенность веб-браузеров, мобильных устройств и интернета вещей. Эффективный, поддерживаемый и правильный код желателен во всех случаях, и я твердо убежден, что C++ является подходящим языком для работы *в целом*.

В руках опытного программиста C++ может создавать меньший, более эффективный и более читаемый код, чем любой другой язык системного программирования на планете. Это язык, который стремится к абстракциям с нулевыми издержками. Вы сможете быстро программировать и вместе с простым прямым сопоставлением с аппаратным обеспечением иметь низкоуровневый контроль в любой момент. Программируя на C++, вы стоите на плечах гигантов, которые десятилетиями создавали невероятно мощный и гибкий язык.

Огромным преимуществом изучения C++ является бесплатный доступ к стандартной библиотеке C++, *stdlib*. Stdlib состоит из трех взаимосвязанных частей: контейнеров, итераторов и алгоритмов. Если вы когда-либо писали свой собственный алгоритм быстрой сортировки вручную или программировали системный код и столкнулись с переполнением буфера, висячими указателями, освобождением после использования и двойными освобождениями, то stdlib придется вам по душе. Она предоставляет непревзойденную комбинацию безопасности, правильности и эффективности типов. Кроме того, вам понравится то, насколько компактным и выразительным может быть код.

В основе модели программирования C++ лежит *жизненный цикл объекта*, который дает надежные гарантии того, что ресурсы, используемые программой: файлы, память и сетевые сокеты, освобождаются правильно даже в случае возникновения ошибок. При эффективном использовании исключения могут убрать из кода большое количество помех, связанных с проверкой условий. Кроме того, семантика перемещения/копирования обеспечивает безопасность, эффективность и гибкость для управления владением ресурсами так, как не могли предоставить ранние языки системного программирования, такие как C.

C++ — живой язык. Спустя более 30 лет комитет Международной организации по стандартизации (ISO) по C++продолжает регулярно вносить улучшения в язык. За последнее десятилетие было выпущено несколько обновлений стандарта: C++11, C++14 и C++17 — в 2011, 2014 и 2017 годах соответственно. В 2020 году вышел C++20.

Когда я использую термин *современный* C++, то имею в виду последний релиз C++, который включает в себя функции и парадигмы, представленные в этих дополнениях. Эти обновления внесли серьезные улучшения в язык, усовершенствовав его выразительность, эффективность, безопасность и общее удобство использования. По некоторым параметрам язык никогда не был более популярным, и он не исчезнет в ближайшее время. Если вы решите инвестировать в изучение C++, он будет приносить дивиденды долгие годы.

Об этой книге

Современные программисты на C++ имеют доступ к ряду очень качественных книг, например «Эффективный современный C++» Скотта Мейерса[1] и «Язык программирования C++» Бьёрна Страуструпа, 4-е издание[2]. Однако эти книги написаны для достаточно продвинутых программистов. Доступны также некоторые вводные тексты о C++, но они часто пропускают важные детали, потому что ориентированы на абсолютных новичков в программировании. Опытному программисту непонятно, где можно погрузиться в язык C++.

Я предпочитаю изучать сложные темы осознанно, выстраивая концепции из их основных элементов. Язык C++ имеет пугающую репутацию, потому что его фундаментальные элементы тесно связаны друг с другом, что затрудняет построение полной картины языка. Когда я изучал C++, то изо всех сил пытался сосредоточиться на языке, перескакивая от книг к видео и измученным коллегам. Поэтому и написал такую книгу, которую сам хотел бы иметь пять лет назад.

Кому будет интересна эта книга?

Эта книга предназначена для программистов среднего и продвинутого уровня, уже знакомых с основными концепциями программирования. Если у вас нет опыта в программировании *систем*, ничего страшного. Опытным программистам приложений издание также будет полезно.

---

[1] *Мейерс С.* М45 Эффективный и современный C++: 42 рекомендации по использованию C++ 11 и C++14 / Пер. с англ. — М.: ООО «ИЛ. Вильямс», 2016. — 304 с. — *Примеч. ред.*

[2] На русском языке можно найти перевод специального издания книги: *Страуструп Б.* Язык программирования C++. Специальное издание. М: Бином, 2020. — 1136 с. — *Примеч. ред.*

ПРИМЕЧАНИЕ

Если вы опытный программист на C или начинающий системный программист, задающийся вопросом, стоит ли вкладывать средства в изучение C++, обязательно прочитайте «Увертюру для C-программистов».

Структура книги

Книга разделена на две части. Часть I охватывает основной язык C++. Здесь не будет истории языка C++ (от старого стиля C++ 98 до современного C++ 11/14/17). Вы изучите непосредственно идиоматический современный C++. Часть II познакомит вас с миром стандартной библиотеки C++ (stdlib), где вы узнаете самые важные понятия.

Часть I. Основы языка C++

- **Глава 1. Создаем и запускаем.** Эта вводная глава поможет настроить среду разработки C++. Вы скомпилируете и запустите свою первую программу и узнаете, как ее отладить.
- **Глава 2. Типы.** Здесь вы познакомитесь с системой типов в C++. Вы узнаете об основных типах — фундаменте, на котором строятся остальные типы. Затем узнаете о старых простых типах данных и полнофункциональных классах, изучите роль конструкторов, инициализации и деструкторов.
- **Глава 3. Ссылочные типы.** В этой главе вы познакомитесь с объектами, которые хранят адреса других объектов в памяти. Эти типы являются краеугольным камнем многих важных паттернов программирования и позволяют создавать гибкий и эффективный код.
- **Глава 4. Жизненный цикл объекта.** Обсуждение инвариантов класса и конструктора продолжается в контексте продолжительности хранения. Деструктор вводится вместе с парадигмой инициализации ресурсов (RAII). Вы узнаете об исключениях и о том, как они реализуют инварианты классов и дополняют RAII. После обсуждения семантики перемещения и копирования вы узнаете, как их реализовать с помощью конструкторов и операторов присваивания.
- **Глава 5. Полиморфизм во время выполнения.** Рассказывает об интерфейсах — концепции программирования, которая позволяет писать код, являющийся полиморфным во время выполнения. Вы изучите базис наследования и композиции объектов, которые лежат в основе использования интерфейсов в C++.
- **Глава 6. Полиморфизм во время компиляции.** В этой главе представлены шаблоны — языковая конструкция, позволяющая писать полиморфный код. Вы также изучите концепции, которые будут добавлены в будущем выпуске C++, и именованные функции преобразования, позволяющие преобразовывать объекты из одного типа в другой.

- **Глава 7. Выражения.** Здесь речь пойдет об операндах и операторах. Хорошо разбираясь в типах, жизненном цикле объектов и шаблонах, вы будете готовы погрузиться в основные компоненты языка C++. Инструкции — первая остановка на этом пути.
- **Глава 8. Инструкции.** В этой главе рассматриваются элементы, составляющие функции. Вы узнаете об операторах выражений, составных операторах, операторах объявлений, операторах итерации и операторах перехода.
- **Глава 9. Функции.** В последней главе части I подробно рассматривается группировка утверждений в единицы работы. Вы узнаете подробности об определениях функций, типах возвращаемых данных, разрешении перегрузки, переменных функциях, переменных шаблонах и указателях функций. Вы также узнаете, как создавать вызываемые пользовательские типы, используя оператор вызова функции и лямбда-выражения. Вы изучите `std::function`, класс, который предоставляет единый контейнер для хранения вызываемых объектов.

Часть II. Библиотеки и фреймворки C++

- **Глава 10. Тестирование.** Эта глава познакомит с удивительным миром фреймворков юнит-тестирования и имитации (мокирования). Вы попрактикуетесь в разработке через тестирование на примере автономной системы вождения, изучая такие фреймворки, как Boost Test, Google Test, Google Mock и другие.
- **Глава 11. Умные указатели.** Здесь описываются специальные служебные классы, которые предоставляет stdlib для управления владением динамическими объектами.
- **Глава 12. Утилиты.** Здесь вы найдете обзор типов, классов и функций, имеющихся в библиотеках stdlib и Boost для решения распространенных проблем программирования. Вы узнаете о структурах данных, числовых функциях и генераторах случайных чисел.
- **Глава 13. Контейнеры.** В этой главе рассматриваются специальные структуры данных из библиотек Boost и stdlib, которые помогают организовать данные. Вы узнаете о контейнерах последовательностей, ассоциативных контейнерах и неупорядоченных ассоциативных контейнерах.
- **Глава 14. Итераторы.** Это интерфейс между контейнерами, о которых вы узнали в предыдущей главе, и строками из следующей главы. Вы узнаете о различных видах итераторов и о том, как их дизайн обеспечивает невероятную гибкость.
- **Глава 15. Строки.** В этой главе рассказывается, как обрабатывать данные на естественном языке в одном семействе контейнеров. Также узнаете о специальных средствах, встроенных в строки, которые позволяют решать общие задачи.
- **Глава 16. Потоки.** Здесь вы познакомитесь с основной концепцией, лежащей в основе операций ввода и вывода. Вы узнаете, как обрабатывать потоки ввода и вывода с помощью форматированных и неформатированных операций,

а также как использовать манипуляторы. Вы узнаете, как читать и записывать данные из файлов и в файлы.

- **Глава 17. Файловые системы.** Здесь вы получите обзор возможностей stdlib для управления файловыми системами. Вы узнаете, как создавать и управлять путями, проверять файлы и каталоги и перечислять структуры каталогов.
- **Глава 18. Алгоритмы.** Эта глава представляет собой краткий справочник по десяткам задач, которые можно легко решить с помощью stdlib. Вы узнаете о впечатляющих возможностях высококачественных алгоритмов.
- **Глава 19. Конкурентность и параллелизм.** В этой главе рассказывается о некоторых простых методах конкурентного программирования, которые являются частью stdlib. Вы узнаете о фьючерсах, мьютексах, условных переменных и атомарности.
- **Глава 20. Сетевое программирование с помощью Boost Asio.** Здесь вы узнаете, как создавать высокопроизводительные программы, которые взаимодействуют по сетям. Вы увидите, как использовать Boost Asio с блокирующим и неблокирующим вводом и выводом.
- **Глава 21. Создание приложений.** Последняя глава завершает книгу обсуждением нескольких важных тем. Вы узнаете о средствах поддержки программ, которые позволяют подключиться к жизненному циклу приложения. Также вы узнаете о Boost ProgramOptions — библиотеке, которая делает написание консольных приложений, принимающих ввод данных пользователем, простым.

ПРИМЕЧАНИЕ

На сайте ccc.codes вы можете найти фрагменты кода из этой книги.

Увертюра для С-программистов

> Артур Дент: «Что с ним не так?»
>
> Хиг Хуртенфлюрст: «Его ноги не подходят под размер его ботинок».
>
> *Дуглас Адамс, «Автостопом по Галактике. Вопль одиннадцатый»*

Это предисловие предназначено для опытных программистов на C, которые сомневаются, стоит ли читать эту книгу. Программисты, не владеющие C, могут эту часть пропустить.

Бьёрн Страуструп разработал C++ из языка программирования C. Хотя C++ не полностью совместим с языком C, хорошо написанные C-программы часто также являются допустимыми C++-программами. Например, каждый пример из «Языка программирования Си»[1] (The C Programming Language) Брайана Кернигана (Brian Kernighan) и Денниса Ритчи (Dennis Ritchie) — допустимая программа на C++.

Одна из главных причин распространенности C в сообществе системного программирования состоит в том, что C позволяет писать на более высоком уровне абстракции, чем программирование на ассемблере. Это приводит к созданию более чистого, менее подверженного ошибкам и проще обслуживаемого кода.

Как правило, системные программисты не хотят платить за удобство программирования, поэтому C придерживается принципа нулевых издержек: *вы не платите за то, что не используете*. Строго типизированная система является ярким примером абстракции с нулевыми издержками. Она используется только во время компиляции для проверки правильности программы. Со временем компиляция типов исчезнет, а в готовом коде сборки не останется никаких следов системы типов.

Будучи потомком C, язык C++ также очень серьезно относится к абстракции без издержек и прямому сопоставлению с аппаратным обеспечением. Это обязательство выходит за рамки возможностей языка C, которые поддерживает C++. Все, что C++ строит поверх C, включая новые языковые возможности, поддерживает эти принципы. Отходы от них сделаны очень осознанно. На самом деле некоторые

---

[1] *Керниган Б., Ритчи Д.* Язык программирования Си / Пер. с англ. 3-е изд., испр. — СПб.: Невский Диалект, 2001.— 352 с. — *Примеч. ред.*

функции C++ несут еще меньше издержек, чем соответствующий код C. Ключевое слово `constexpr` является одним из таких примеров. Оно указывает компилятору вычислить выражение во время компиляции (если это возможно), как показано в листинге 1.

Листинг 1. Программа, демонстрирующая constexpr

```
#include <cstdio>

constexpr int isqrt(int n) {
  int i=1;
  while (i*i<n) ++i;
  return i-(i*i!=n);
}

int main() {
  constexpr int x = isqrt(1764); ❶
  printf("%d", x);
}
```

Функция `isqrt` вычисляет квадратный корень аргумента `n`. Начиная с `1`, функция увеличивает локальную переменную `i` до тех пор, пока `i*i` не станет больше или равно `n`. Если `i*i == n`, возвращается `i`; в противном случае возвращается `i-1`. Обратите внимание, что вызов `isqrt` имеет буквальное значение, поэтому компилятор теоретически может вычислить результат, который будет принимать только одно значение ❶.

Компиляция листинга 1 в GCC 8.3 для x86-64 с `-O2` приведет к сборке из листинга 2.

Листинг 2. Сборка, созданная после компиляции листинга 1

```
.LC0:
        .string "%d"
main:
        sub rsp, 8
        mov esi, 42 ❶
        mov edi, OFFSET FLAT:.LC0
        xor eax, eax
        call printf
        xor eax, eax
        add rsp, 8
        ret
```

Важным результатом здесь является вторая инструкция в `main` ❶; вместо того чтобы вычислять квадратный корень из `1764` во время выполнения, компилятор оценивает его и выводит инструкции для обработки `x` как `42`. Конечно, можно вычислить квадратный корень с помощью калькулятора и вставить результат вручную, но использование `constexpr` имеет много преимуществ. Этот подход может снизить вероятность многих ошибок, связанных с ручным копированием и вставкой, и сделать код более выразительным.

ПРИМЕЧАНИЕ

Если вы не знакомы со сборкой x86, почитайте второе издание «Искусства ассемблера» Рэндалла Хайда (The Art of Assembly Language, Randall Hyde) и «Язык профессиональной сборки» Ричарда Блума (Professional Assembly Language, Richard Blum).

Обновление до Super C

Современные компиляторы C++ учтут большинство ваших навыков программирования на C. Это облегчает использование некоторых тактических изысков, которые предоставляет язык C++, но в то же время обходит более глубокие темы языка. Такой стиль C++ — назовем его *Super* C — важно обсудить по нескольким причинам. Во-первых, опытные программисты на C могут сразу получить выгоду от применения простых тактических понятий C++ к своим программам. Во-вторых, Super C *не* является идиоматическим C++. Простое добавление ссылок и экземпляров `auto` в C-программы может сделать код более надежным и читаемым, но для того, чтобы в полной мере воспользоваться этими преимуществами, стоит изучить другие концепции. В-третьих, в некоторых строгих средах (например, встроенное программное обеспечение, некоторые ядра операционных систем и гетерогенные вычисления) доступные цепочки инструментов имеют неполную поддержку C++. В таких ситуациях можно извлечь выгоду по крайней мере из некоторых идиом C++ и Super C, скорее всего, будет поддерживаться. В этом разделе рассматриваются некоторые концепции Super C, которые можно тут же применить к своему коду.

ПРИМЕЧАНИЕ

Некоторые C-поддерживаемые конструкции не будут работать в C++. Смотрите раздел ссылок на сайте этой книги: ccc.codes.

Перегрузка функций

Рассмотрим следующие функции преобразования из стандартной библиотеки C:

```
char* itoa(int value, char* str, int base);
char* ltoa(long value, char* buffer, int base);
char* ultoa(unsigned long value, char* buffer, int base);
```

Эти функции служат одной и той же цели: преобразуют целочисленный тип в строку в стиле C. В языке C каждая функция должна иметь уникальное имя. Но в C++ функции могут совместно использовать имена при наличии различий в их аргументах; это называется *перегрузкой функции* (*function overloading*). Перегрузку функций можно использовать для создания своих собственных функций преобразования, как показано в листинге 3.

Листинг 3. Вызов перегруженных функций

```
char* toa(int value, char* buffer, int base) {
  --пропуск--
}

char* toa(long value, char* buffer, int base)
  --пропуск--
}

char* toa(unsigned long value, char* buffer, int base) {
  --пропуск--
}

int main() {
  char buff[10];
  int a = 1; ❶
  long b = 2; ❷
  unsigned long c = 3; ❸
  toa(a, buff, 10);
  toa(b, buff, 10);
  toa(c, buff, 10);
}
```

Тип данных первого аргумента в каждой из функций отличается, поэтому компилятор C++ имеет достаточно информации из аргументов, переданных в `toa`, для вызова правильной функции. Каждый вызов `toa` является уникальной функцией. Здесь были созданы переменные `a` ❶, `b` ❷ и `c` ❸, которые представляют собой различные типы объектов `int`, которые соответствуют одной из трех функций `toa`. Это удобнее, чем определять отдельно названные функции, потому что достаточно запомнить одно имя, а компилятор сам определит, какую функцию вызывать.

Ссылки

Указатели являются важной особенностью C (и, как следствие, системного программирования в целом). Они позволяют эффективно обрабатывать большие объемы данных, передавая адреса в памяти вместо фактических данных. Указатели одинаково важны и для C++, но в него были добавлены дополнительные функции безопасности, которые защищают от нулевых разыменований и непреднамеренных переназначений указателей. *Ссылки* значительно улучшают работу с указателями. Они похожи на указатели, но имеют некоторые ключевые отличия. Синтаксически ссылки отличаются от указателей по двум важным причинам. Во-первых, они должны быть объявлены знаком `&` вместо `*`, как показано в листинге 4.

Листинг 4. Код, показывающий, как объявлять функции с указателями и ссылками

```
struct HolmesIV {
  bool is_sentient;
  int sense_of_humor_rating;
};
void make_sentient(HolmesIV*); // Принимает указатель на HolmesIV HolmesIV
void make_sentient(HolmesIV&); // Принимает ссылку на HolmesIV
```

Во-вторых, взаимодействовать с членами нужно, используя оператор точки `.`, а не оператор стрелки `->`, как показано в листинге 5.

Листинг 5. Программа с операторами точки и стрелки

```
void make_sentient(HolmesIV* mike) {
  mike->is_sentient = true;
}

void make_sentient(HolmesIV& mike) {
  mike.is_sentient = true;
}
```

Под капотом ссылки эквивалентны указателям, потому что также являются абстракцией с нулевыми издержками. Компилятор в итоге выдает похожий код. Чтобы продемонстрировать это, рассмотрим результаты компиляции функций `make_sentient` в GCC 8.3 для x86-64 с `-O2`. В листинге 6 содержится сборка, созданная путем компиляции листинга 5.

Листинг 6. Сборка, сгенерированная путем компиляции листинга 5

```
make_sentient(HolmesIV*):
        mov     BYTE PTR [rdi], 1
        ret
make_sentient(HolmesIV&):
        mov     BYTE PTR [rdi], 1
        ret
```

Однако во время компиляции ссылки обеспечивают некоторую безопасность по сравнению с обычными указателями, потому что вообще-то они не могут быть нулевыми.

С помощью указателей можно добавить проверку `nullptr` в целях безопасности. Например, проверку можно провести в `make_sentient`, как показано в листинге 7.

Листинг 7. Рефакторинг функции make_sentient из листинга 5: теперь она выполняет проверку nullptr

```
void make_sentient(HolmesIV* mike) {
  if(mike == nullptr) return;
  mike->is_sentient = true;
}
```

Такая проверка не требуется при получении ссылки; однако это не означает, что ссылки всегда корректны. Рассмотрим следующую функцию:

```
HolmesIV& not_dinkum() {
  HolmesIV mike;
  return mike;
}
```

Функция `not_dinkum` возвращает ссылку, которая точно не равна `null`. Но она указывает на ненужную информацию в памяти (вероятно, в возвращаемом кадре

стека not_dinkum). Никогда так не делайте. Это приведет к беде, также известной как *неопределенное поведение во время выполнения* (*undefined runtime behavior*): может произойти сбой, ошибка или что-то совершенно неожиданное. Еще одной особенностью безопасности ссылок является то, что они не могут быть переопределены. Другими словами, после инициализации ссылки ее нельзя изменить так, чтобы она указывала на другой адрес в памяти, как показано в листинге 8.

Листинг 8. Пример, показывающий, что ссылки не могут быть повторно установлены

```
int main() {
  int a = 42;
  int& a_ref = a; ❶
  intb = 100;
  a_ref = b; ❷
}
```

a_ref была объявлена как ссылка на int a ❶. Не существует способа переопределить a_ref так, чтобы она указывала на другой int. Можно попробовать переопределить a с помощью operator= ❷, но это фактически устанавливает значение a равным значению b вместо установки a_ref в качестве ссылки на b. После запуска фрагмента и a, и b равны 100, а a_ref по-прежнему указывает на a. В листинге 9 приведен эквивалентный код с использованием указателей.

Листинг 9. Программа, эквивалентная листингу 8, с указателями

```
int main() {
  int a = 42;
  int* a_ptr = &a; ❶
  intb = 100;
  *a_ptr = b; ❷
}
```

Здесь указатель объявляется с помощью * вместо & ❶. Вы присваиваете значение b памяти, на которую указывает a_ptr ❷. При использовании ссылок не потребуется никаких декораций слева от знака равенства. Но если опустить * в *a_ptr, компилятор пожалуется на попытку присвоить int типу указателя.

Ссылки — это просто указатели с дополнительными мерами предосторожности и небольшим количеством синтаксического сахара. Размещая ссылку слева от знака равенства, вы устанавливаете целевое значение указателя, равное правой стороне знака равенства.

Инициализация с помощью auto

В C часто требуется повтор информации о типе более одного раза. В C++ можно выразить информацию о типе переменной только один раз, используя ключевое слово auto. Компилятор будет знать тип переменной, потому что он знает тип зна-

чения, используемого для инициализации переменной. Рассмотрим следующие инициализации переменных в C++:

```
int x = 42;
auto y = 42;
```

Здесь и `x`, и `y` имеют тип `int`. Вы можете удивиться, узнав, что компилятор может определить тип `y`, но учтите, что 42 — это целочисленный литерал. При использовании `auto` компилятор выводит тип справа от знака равенства `=` и устанавливает такой же тип переменной. Поскольку целочисленный литерал имеет тип `int`, в этом примере компилятор определяет, что тип `y` также является типом `int`. Для такого простого примера это не кажется большим преимуществом, но давайте рассмотрим возможность инициализации переменной возвращаемым значением функции, как показано в листинге 10.

Листинг 10. Программа, инициализирующая переменную возвращаемым значением функции

```
#include <cstdlib>

struct HolmesIV {
  --пропуск--
};

HolmesIV* make_mike(int sense_of_humor) {
  --пропуск--
}

int main() {
  auto mike = make_mike(1000);
  free(mike);
}
```

Ключевое слово `auto` проще прочитать, и оно легче поддается рефакторингу, чем явное объявление типа переменной. Если вы свободно используете `auto` при объявлении функции, понадобится меньше работы для изменения типа возвращаемого значения `make_mike`. Использование `auto` усиливается более сложными типами, такими как типы, связанные с кодом, загружаемым из шаблона `stdlib`. Ключевое слово `auto` заставляет компилятор выполнять за вас всю работу по выводу типов.

ПРИМЕЧАНИЕ

В `auto` также можно добавить квалификаторы `const`, `volatile`, `&` и `*`.

Пространства имен и неявный typedef для struct, union и enum

C++ обрабатывает теги типа как неявные имена `typedef`. В C для использования `struct`, `union` или `enum` нужно назначить имя созданному типу с помощью ключевого слова `typedef`. Например:

```
typedef struct Jabberwocks {
  void* tulgey_wood;
  int is_galumphing;
} Jabberwock;
```

В кругах C++ над таким кодом можно только посмеяться. Поскольку ключевое слово `typedef` может быть неявным, C++ позволяет вместо него объявлять тип `Jabberwock` следующим образом:

```
struct Jabberwock {
  void* tulgey_wood;
  int is_galumphing;
};
```

Это удобнее и экономит время при наборе текста. Что произойдет, если также нужно определить функцию `Jabberwock`? Вообще, этого делать не стоит, потому что повторное использование одного и того же имени для типа данных и функции может породить путаницу. Но если без этого не обойтись, C++ позволяет объявлять пространство имен для создания различных областей действия для идентификаторов. Это помогает поддерживать чистоту пользовательских типов и функций, как показано в листинге 11.

Листинг 11. Использование пространств имен для устранения неоднозначности функций и типов с одинаковыми именами

```
#include <cstdio>

namespace Creature { ❶
  struct Jabberwock {
    void* tulgey_wood;
    int is_galumphing;
  };
}
namespace Func { ❷
  void Jabberwock() {
    printf("Burble!");
  }
}
```

В этом примере `struct Jabberwock` и функция `Jabberwock` теперь живут вместе в гармонии. Поместив каждый элемент в свое собственное пространство имен — `struct` в пространство имен `Creature` ❶ и функцию в пространство имен `Func`❷, — можно определить, какой из `Jabberwock` имеется в виду. Достичь такого устранения неоднозначности можно несколькими способами. Самое простое — определить имя по его пространству имен, например:

```
Creature::Jabberwock x;
Func::Jabberwock();
```

Также можно использовать директиву `using` для импорта всех имен в пространстве имен, поэтому больше не нужно будет использовать полное имя элемента. В листинге 12 используется пространство имен `Creature`.

Листинг 12. Использование пространства имен для ссылки на тип в пространстве имен Creature

```
#include <cstdio>

namespace Creature {
  struct Jabberwock {
    void* tulgey_wood;
    int is_galumphing;
  };
}

namespace Func {
  void Jabberwock() {
    printf("Burble!");
  }
}

using namespace Creature;  ❶

int main() {
  Jabberwock x; v ❷
  Func::Jabberwock();
}
```

`using namespace` ❶ позволяет опустить квалификацию `namespace` ❷. Но вам все равно нужен спецификатор в `Func::Jabberwock`, потому что он не является частью `Creature namespace`.

Использование `namespace` — пример идиоматического C++ и абстракции с нулевыми издержками. Как и остальные идентификаторы типа, `namespace` стирается компилятором при выдаче кода сборки. В больших проектах это невероятно полезно для разделения кода в разных библиотеках.

Смешение объектных файлов C и C++

Коды C и C++ могут мирно сосуществовать, если проявить предусмотрительность. Иногда компилятору C необходимо связать объектные файлы, выдаваемые компилятором C++ (и наоборот). Это реально, хотя и потребует немного усилий.

Со связыванием файлов возникают две проблемы. Во-первых, соглашения о вызове в кодах C и C++ могут потенциально не совпадать. Например, протоколы установки стека и регистров при вызове функции могут отличаться. Эти соглашения о вызове являются несоответствиями на уровне языка и обычно не связаны с тем, как вы написали функции. Во-вторых, компиляторы C++выдают символы, отличные от символов компиляторов C. Иногда редактор связей должен идентифицировать объект по имени. Компиляторы C++ помогают декорировать объект, связывая строку, называемую *декорированным именем* (*decorated name*), с объектом. Из-за перегрузок функций, соглашений о вызове и использования `namespace` компилятор должен кодировать дополнительную информацию о функции, помимо ее имени, посредством декорирования. Это сделано для того, чтобы редактор связей мог одно-

значно идентифицировать функцию. К сожалению, не существует стандарта того, как это оформление происходит в C++ (именно поэтому необходимо использовать одну и ту же цепочку инструментов и настройки при связывании между единицами перевода). Редакторы связей C ничего не знают о декорировании имен C++, что может вызвать проблемы, если декорирование не подавляется при ссылках на код C в C++ (и наоборот).

Исправить это легко. Оберните код, который нужно скомпилировать, в стиле C, используя оператор `extern "C"`, как показано в листинге 13.

Листинг 13. Использование связей в стиле C

```
// header.h
#ifdef __cplusplus
extern "C" {
#endif
void extract_arkenstone();

struct MistyMountains {
  int goblin_count;
};
#ifdef __cplusplus
}
#endif
```

Этот заголовок может быть разделен на код C и C++ . Такой подход работает, потому что `__cplusplus` — это специальный идентификатор, который определяет компилятор C++ (но компилятор C этого не делает). Соответственно компилятор C видит код в листинге 14 после завершения предварительной обработки. В листинге 14 показан оставшийся код.

Листинг 14. Код, оставшийся после обработки препроцессором листинга 13 в среде C

```
void extract_arkenstone();

struct MistyMountains {
  int goblin_count;
};
```

Это простой C-заголовок. Код между операторами `#ifdef __cplusplus` удаляется во время предварительной обработки, поэтому обертка `extern "C"` остается незамеченной. Для компилятора C++ `__cplusplus` определен в `header.h`, поэтому он видит содержимое листинга 15.

Листинг 15. Код, оставшийся после обработки препроцессором листинга 13 в среде C++

```
extern "C" {
  void extract_arkenstone();

  struct MistyMountains {
    int goblin_count;
  };
}
```

И extract_arkenstone, и MistyMountains теперь заключены в extern "C", поэтому компилятор знает, как использовать связь в стиле С. Теперь источник С может вызывать скомпилированный код С++, а источник С++ может вызывать скомпилированный код С.

Темы С++

В этом разделе вы познакомитесь с некоторыми основными темами, которые делают С++ лучшим языком системного программирования. Не беспокойтесь о деталях. Задача следующих подразделов — разжечь интерес.

Краткое изложение идей и повторное использование кода

Хорошо разработанный код С++элегантен и компактен. Рассмотрим переход от ANSI-С к современному С++ в следующей простой операции: перебор элементов в некотором массиве v с n элементами, как показано в листинге 16.

Листинг 16. Программа, показывающая несколько способов перебора массива

```
#include <cstddef>

int main() {
  const size_t n{ 100 };
  int v[n];

  // ANSI-C
  size_t i;
  for (i=0; i<n; i++) v[i]=0; ❶
  // C99
  for (size_t i=0; i<n; i++) v[i]=0; ❷

  // C++17
  for (auto& x : v) x = 0; ❸
}
```

В этом фрагменте кода показаны различные способы объявления циклов в ANSI-С, С99 и С++. Индексная переменная i в примерах ANSI-С ❶ и С99 ❷ является вспомогательной для того, что вы пытаетесь достичь, а именно для доступа к каждому элементу v. В версии С++ ❸ используется цикл for, *основанный на диапазоне*, который перебирает диапазон значений в v, скрывая детали того, как достигается итерация. Как и многие абстракции с нулевыми издержками в С++, эта конструкция позволяет сосредоточиться на значении, а не на синтаксисе. Циклы for на основе диапазона работают со многими типами, и можно даже заставить их работать с пользовательскими типами.

Что касается пользовательских типов, то они позволяют выражать идеи непосредственно в коде. Предположим, нужно сконструировать функцию navigate_to, кото-

рая сообщает гипотетическому роботу, что нужно перейти к некоторому положению с заданными координатами `x` и `y`. Рассмотрим следующую функцию-прототип:

```
void navigate_to(double x, double y);
```

Что такое `x` и `y`? Что за системы они представляют? Пользователь должен прочитать документацию (или, возможно, исходный код), чтобы это узнать. Сравните код выше с его улучшенной версией:

```
struct Position{
--пропуск--
};
void navigate_to(const Position& p);
```

Функция стала намного понятнее. Нет никакой двусмысленности в том, что принимает `navigate_to`. Пока существует правильно сконструированный экземпляр `Position`, вы точно знаете, как вызывать `navigate_to`. Беспокоиться о единицах, конверсиях и т. д. теперь стоит тому, кто создает класс `Position`.

К подобной ясности также можно приблизиться в C99/C11, используя указатель `const`, но C++ также делает возвращаемые типы компактными и выразительными. Предположим, нужно написать сопутствующую функцию `get_position` для робота, которая — как вы уже догадались — получает позицию. В C для этого существуют два варианта, как показано в листинге 17.

Листинг 17. API в стиле C для возврата пользовательского типа

```
Position* get_position(); ❶
void get_position(Position* p); ❷
```

В первом варианте вызывающая сторона отвечает за очистку возвращаемого значения ❶, которое, вероятно, подверглось динамическому распределению (хотя это неочевидно из кода). Вызывающая сторона ответственна за то, чтобы где-то выделить память для `Position` и передать полученный экземпляр в `get_position` ❷. Последний подход выражен в стиле C, но тут язык начинает мешать: вы просто пытаетесь получить объект позиции, при этом нужно беспокоиться о том, отвечает ли вызывающая или вызванная функция за выделение и освобождение памяти. C++ позволяет делать все это лаконично, возвращая пользовательские типы непосредственно из функций, как показано в листинге 18.

Листинг 18. Возвращение пользовательского типа по значению в C++

```
Position❶ get_position() {
  --пропуск--
}
void navigate() {
  auto p = get_position(); ❷
  // с этого момента можно использовать p
  --пропуск--
}
```

Поскольку `get_position` возвращает значение ❶, компилятор может аннулировать копию, поэтому создается впечатление, что автоматическая переменная `Position` была создана непосредственно ❷; не было издержек времени выполнения. Функционально вы находитесь на территории, очень похожей на проход в стиле C, как показано в листинге 17.

Стандартная библиотека C++

Стандартная библиотека C++ (stdlib) является основной причиной перехода из C. Она содержит высокопроизводительный общий код, который гарантированно доступен по умолчанию и при этом соответствует стандартам. Три важнейших компонента stdlib — это контейнеры, итераторы и алгоритмы.

Контейнеры — это структуры данных. Они отвечают за хранение последовательностей объектов. Они правильные, безопасные и (обычно) как минимум настолько же эффективные, как и то, что вы могли бы выполнить вручную. Это означает, что написать собственные версии этих контейнеров было бы сложнее и они не получились бы лучше, чем существующие контейнеры stdlib. Контейнеры четко разделены на две категории: *контейнеры последовательности* (*sequence containers*) и *ассоциативные контейнеры* (*associative containers*). Контейнеры последовательности концептуально похожи на массивы; они обеспечивают доступ к последовательностям элементов. Ассоциативные контейнеры содержат пары ключ/значение, поэтому элементы в контейнерах можно искать по ключу.

Алгоритмы stdlib являются функциями общего назначения для основных задач программирования, таких как подсчет, поиск, сортировка и преобразование. Подобно контейнерам, алгоритмы stdlib чрезвычайно качественны и широко применяются. Пользователям очень редко приходится реализовывать свою собственную версию, а использование алгоритмов stdlib значительно повышает производительность труда программиста, безопасность кода и удобочитаемость.

Итераторы связывают контейнеры с алгоритмами. Во многих случаях использования алгоритма stdlib данные, с которыми нужно работать, находятся в контейнере. Контейнеры предоставляют итераторы для обеспечения общего интерфейса, а алгоритмы используют итераторы, не позволяя программистам (включая разработчиков stdlib) реализовывать собственный алгоритм для каждого типа контейнера.

В листинге 19 показано, как отсортировать контейнер значений, используя несколько строк кода.

В фоновом режиме выполняется большое количество вычислений, но итоговый код компактен и выразителен. Сначала инициализируется контейнер `std::vector` ❶. *Векторы* — это динамические массивы stdlib. *Скобки инициализатора* (*initializer braces*) ({0, 1,...}) устанавливают начальные значения, содержащиеся в `x`. Доступ к элементам вектора можно получить так же, как к элементам массива, используя скобки (`[]`) и порядковый номер. Эта техника используется, чтобы установить первый элемент равным `21` ❷. Поскольку векторные массивы имеют динамический

размер, можно добавлять значения к ним, используя метод `push_back` ❸. Кажущийся магическим вызов `std::sort` демонстрирует мощь алгоритмов в `stdlib` ❹. Методы `x.begin()` и `x.end()` возвращают итераторы, которые `std::sort` использует для сортировки `x` на месте. Алгоритм сортировки отделен от вектора с помощью итераторов.

Листинг 19. Сортировка контейнера значений с использованием stdlib

```
#include <vector>
#include <algorithm>
#include <iostream>

int main() {
  std::vector<int> x{ 0, 1, 8, 13, 5, 2, 3 }; ❶
  x[0] = 21;  ❷
  x.push_back(1); ❸
  std::sort(x.begin(), x.end()); ❹
  std::cout << "Printing " << x.size() << " Fibonacci numbers.\n"; ❺
  for (auto number : x) {
    std::cout << number << std::endl; ❻
  }
}
```

Благодаря итераторам можно использовать другие контейнеры в stdlib аналогичным образом. Например, вместо вектора может быть использован список (двусвязный список в stdlib). Поскольку `list` также предоставляет итераторы с помощью методов `.begin()` и `.end()`, `sort` вызывается таким же образом для итераторов списка.

Кроме того, в листинге 19 используются потоки ввода/вывода. *Потоки ввода/вывода* (*iostreams*) — это механизм stdlib для выполнения буферизованного ввода и вывода. Оператор сдвига влево (`<<`) используется для потоковой передачи значения `x.size()` (количество элементов в `x`), некоторых строковых литералов и элемента Фибоначчи `number` в `std::cout`, который инкапсулирует стандартный вывод ❺ ❻. Объект `std::endl` является манипулятором ввода-вывода, который записывает `\n` и очищает буфер, гарантируя, что весь поток записывается в стандартный вывод перед выполнением следующей инструкции.

Теперь просто представьте полосу с препятствиями, которую нужно пройти, чтобы написать эквивалентную программу на C, и вы поймете, почему stdlib — такой ценный инструмент.

Лямбда-выражения

Лямбда-выражения (их также называют *анонимными функциями*) являются еще одной мощной языковой функцией, улучшающей локальность кода. В некоторых случаях нужно передавать указатели на функции, чтобы использовать указатель в качестве цели для вновь созданного потока или выполнять какое-либо преобразование для каждого элемента последовательности. Обычно для этого неудобно определять новую одноразовую функцию. Вот здесь и появляются лямбда-выражения. Лямб-

да-выражение — это новая настраиваемая функция, *определяемая в соответствии с другими параметрами вызова*. Рассмотрим следующую однострочную функцию, которая вычисляет количество четных чисел в x:

```
auto n_evens = std::count_if(x.begin(), x.end(),
                             [] (auto number) { return number % 2 == 0; });
```

Этот фрагмент использует алгоритм `count_if` из stdlib для подсчета четных чисел в `x`. Первые два аргумента для `std::count_if` соответствуют `std::sort`; это итераторы, которые определяют диапазон, в котором будет работать алгоритм. Третий аргумент — лямбда-выражение. Обозначения, вероятно, выглядят немного незнакомыми, но довольно просто понять суть:

```
[ввод] (параметры) { тело }
```

Ввод содержит любые объекты, которые нужны, из области, где определяется лямбда-выражение для выполнения вычислений в теле. *Параметры* определяют имена и типы параметров, вызова с которыми ожидает лямбда-выражение. *Тело* содержит любые вычисления, которые нужно выполнить при вызове. Оно может возвращать или не возвращать значение. Компилятор выведет прототип функции на основе подразумеваемых типов.

В приведенном выше вызове `std::count_if` лямбда-код не нуждался во вводе каких-либо переменных. Вся необходимая информация принимается за один аргумент `number`. Поскольку компилятор знает тип элементов, содержащихся в `x`, вы объявляете тип `number` с помощью `auto`, чтобы компилятор мог определить его самостоятельно. Лямбда-выражение вызывается для каждого элемента `x`, переданного в качестве параметра `number`. В теле лямбда-выражение возвращает `true` только тогда, когда `number` делится на 2, поэтому в `number` включаются только четные числа.

Лямбда-выражений в С нет, и реконструировать их невозможно. Нужно будет объявлять отдельную функцию каждый раз, когда необходим функциональный объект. Невозможно передать объекты в функцию таким же образом.

Обобщенное программирование с помощью шаблонов

Обобщенное программирование — это написание за один раз кода, работающего с разными типами, вместо повторения одного и того же кода, когда копируется и вставляется каждый новый поддерживаемый тип. В языке С++ для создания обобщенного кода используются *шаблоны*. Шаблоны — это особый тип параметра, который указывает компилятору на то, что нужно предоставить широкий диапазон возможных типов.

Вы уже знакомы с шаблонами: все контейнеры stdlib используют их. По большей части тип объектов в этих контейнерах не имеет значения. Например, логика определения количества элементов в контейнере или возврата его первого элемента не зависит от типа элемента.

Предположим, вы хотите написать функцию, которая складывает три числа одного типа. Функция должна принимать любой складываемый тип. В C++ — это простая обобщенная задача программирования, которую можно решить непосредственно с помощью шаблонов (листинг 20).

Листинг 20. Использование шаблонов для создания обобщенной функции сложения

```
template <typename T>
T add(T x, T y, T z) { ❶
  return x + y + z;
}

int main() {
  auto a = add(1, 2, 3);       // a имеет тип int
  auto b = add(1L, 2L, 3L);    // b имеет тип long
  auto c = add(1.F, 2.F, 3.F); // c имеет тип float
}
```

При объявлении `add` ❶ вам не нужно знать `T`. Вам нужно только знать, что все параметры и возвращаемое значение имеют тип `T` и что `T` является слагаемым типом. Когда компилятор обнаруживает вызов `add`, он выводит `T` и генерирует сделанную на заказ функцию от вашего имени. Это серьезное повторное использование кода!

Инварианты классов и управление ресурсами

Возможно, самое большое нововведение C++ в системное программирование — это *жизненный цикл объекта*. Эта концепция берет свое начало в C, где объекты имеют различную продолжительность хранения в зависимости от способа объявления их в коде.

C++ основывается на этой модели управления памятью с помощью конструкторов и деструкторов. Эти специальные функции являются методами, которые принадлежат *пользовательским типам*. Пользовательские типы — это основные строительные блоки приложений C++. Можете представить их как объекты структуры, которые также могут иметь функции.

Конструктор объекта вызывается сразу после начала хранения объекта, а деструктор вызывается непосредственно перед окончанием хранения. И конструктор, и деструктор являются функциями без возвращаемого значения и с тем же именем, что и класс, в котором они содержатся. Чтобы объявить деструктор, добавьте `~` в начало имени класса, как показано в листинге 21.

Первый метод в `Hal` — это *конструктор* ❶. Он создает объект `Hal` и устанавливает его *инварианты класса*. Инварианты — это особенности класса, которые не изменяются после создания. С некоторой помощью компилятора и среды выполнения программист решает, что представляют собой инварианты класса, и обеспечивает их применение в коде. В этом случае конструктор устанавливает инвариант — версию, которая равна 9000. Второй метод — это *деструктор* ❷. `Hal` выводит `"Stop, Dave."`

в консоль всякий раз перед собственным уничтожением. (Заставить `Hal` спеть «Дейзи Белл» — упражнение для читателя в конце главы.)

Листинг 21. Класс Hal, содержащий конструктор и деструктор

```
#include <cstdio>

struct Hal {
  Hal() : version{ 9000 } { // Конструктор ❶
    printf("I'm completely operational.\n");
  }
  ~Hal() { // Деструктор ❷
    printf("Stop, Dave.\n");
  }
  const int version;
};
```

Компилятор обеспечивает автоматический вызов конструктора и деструктора для объектов со статической и локальной длительностью хранения. Для объектов с динамической продолжительностью хранения используются ключевые слова `new` и `delete` на замену `malloc` и `free`, как показано в листинге 22.

Листинг 22. Программа, которая создает и уничтожает объект Hal

```
#include <cstdio>

struct Hal {
--пропуск--
};

int main() {
  auto hal = new Hal{}; // Память выделяется, затем вызывается конструктор
  delete hal;           // Вызывается деструктор, затем память освобождается
}
--------------------------------------------------------------------------------
I'm completely operational.
Stop, Dave.
```

Если по какой-либо причине конструктор не может достичь нормального состояния, то обычно он выдает *исключение*. Как программист на C вы могли иметь дело с исключениями, когда использовались некоторые API операционной системы (например, Windows Structured Exception Handling). Когда генерируется исключение, стек разматывается до тех пор, пока не будет найден обработчик исключения, после чего программа восстанавливается. Разумное использование исключений может привести к очистке кода, потому что проверять наличие ошибок стоит только в тех случаях, когда это имеет смысл. C++ имеет поддержку исключений на уровне языка, как показано в листинге 23.

Можно поместить код, который может вызвать исключение, в блок сразу после `try` ❶. Если в какой-то момент выдается исключение, стек будет разматываться (разрушая любые объекты, выходящие за пределы области видимости) и запускать

любой код, размещенный после выражения `catch` ❷. Если исключение не выдается, код в `catch` никогда не выполняется.

Листинг 23. Блок try-catch

```
#include <exception>

try {
  // Код, который может привести к std::exception ❶
} catch (const std::exception &e) {
  // Восстановление программы в этом месте ❷
}
```

Конструкторы, деструкторы и исключения тесно связаны с другой основной темой C++, связывающей жизненный цикл объекта с ресурсами, которыми он владеет. Это называется концепцией распределения ресурсов при инициализации (RAII) (или *получением конструктора — освобождением деструктора* (*constructor acquires, destructor releases*)). Рассмотрим класс C++ в листинге 24.

Листинг 24. Класс File

```
#include <system_error>
#include <cstdio>

struct File {
  File(const char* path, bool write) { ❶
    auto file_mode = write ? "w" : "r"; ❷
    file_pointer = fopen(path, file_mode); ❸
    if (!file_pointer) throw std::system_error(errno, std::system_category()); ❹
  }
  ~File() {
    fclose(file_pointer);
  } FILE* file_pointer;
};
```

Конструктор `File` ❶ принимает два параметра. Первый параметр соответствует пути к файлу, а второй является логическим значением, соответствующим тому, должен ли файл быть открыт для записи (`true`) или чтения (`false`). Значение этого параметра устанавливает `file_mode` ❷ через *тернарный оператор* `? :`. Тернарный оператор вычисляет логическое выражение и возвращает одно из двух значений в зависимости от полученного значения. Например:

```
x ? val_if_true : val_if_false
```

Если логическое выражение `x` истинно, `val_if_true` является значением выражения. Если значение `x` ложно, вместо этого используется значение `val_if_false`.

В фрагменте кода конструктора `File` в листинге 24 конструктор пытается открыть файл в пути `path` с доступом для чтения/записи ❸. Если что-то пойдет не так, при вызове конструктора в `file_pointer` будет передано значение `nullptr`, специальное значение C++, которое похоже на `0`. Если это происходит, генерируется

system_error ❹. system_error — это просто объект, который инкапсулирует детали системной ошибки. Если file_pointer имеет значение, отличное от nullptr, его можно использовать. Это инвариант данного класса.

Теперь рассмотрим программу из листинга 25, в которой используется File.

Листинг 25. Программа, использующая класс File

```
#include <cstdio>
#include <system_error>
#include <cstring>

struct File {
--пропуск--
};

int main() {
  { ❶
    File file("last_message.txt", true); ❷
    const auto message = "We apologize for the inconvenience.";
    fwrite(message, strlen(message), 1, file.file_pointer);
  } ❸
  // last_message.txt закрывается здесь!
  {
    File file("last_message.txt", false); ❹
    char read_message[37]{};
    fread(read_message, sizeof(read_message), 1, file.file_pointer);
    printf("Read last message: %s\n", read_message);
  }
}
--------------------------------------------------------------------------------
We apologize for the inconvenience.
```

Фигурные скобки ❶ ❸ определяют область видимости. Поскольку первый file находится в этой области, область видимости определяет время жизни file. Как только конструктор вернет ❷, вы будет знать, что file.file_pointer является допустимым благодаря инварианту класса; основываясь на дизайне конструктора File, известно, что file.file_pointer должен быть действителен в течение всего времени существования объекта File. С помощью fwrite создается сообщение. Нет необходимости явно вызывать fclose, потому что срок действия file истекает и деструктор очищает file.file_pointer самостоятельно ❸. file открывается снова, но на этот раз для чтения ❹. Как только конструктор вернет значение, вы будете знать, что *last_message.txt* был успешно открыт и, значит, можно продолжить чтение в read_message. После вывода сообщения вызывается деструктор file и file.file_pointer снова очищается.

Иногда необходимо иметь гибкость динамического выделения памяти одновременно с преимуществами жизненного цикла объекта C++. Это гарантирует, что память не будет потеряна или случайно «использована после освобождения». Именно в этом заключается роль *умных указателей*, которые управляют жизненным циклом дина-

мических объектов через модель владения. Как только умный указатель перестает владеть динамическим объектом, объект разрушается.

Одним из таких умных указателей является `unique_ptr`, который моделирует исключительное владение. Листинг 26 показывает его основное применение.

Листинг 26. Программа, использующая unique_ptr

```
#include <memory>

struct Foundation{
  const char* founder;
};

int main() {
  std::unique_ptr second_foundation{ new Foundation{} }; ❶
  // Получите доступ к переменной-члену founder подобно указателю:
  second_foundation->founder = "Wanda";
} ❷
```

Память динамически выделяется для `Foundation`, и результирующий указатель `Foundation*` передается в конструктор `second_foundation` с помощью скобок ❶. `second_foundation` имеет тип `unique_ptr`, который является просто объектом RAII, оберткой для динамического `Foundation`. Когда `second_foundation` уничтожается ❷, динамический `Foundation` уничтожается соответствующим образом.

Умные указатели отличаются от *обычных*, простых указателей, потому что простой указатель является просто адресом памяти. Вся система управления памятью, связанная с адресом, должна быть организована вручную. С другой стороны, умные указатели обрабатывают все эти беспорядочные детали. Оборачивая динамический объект умным указателем, вы можете быть уверены, что память будет очищена надлежащим образом, как только объект станет ненужным. Компилятор знает, что объект больше не нужен, потому что деструктор умного указателя вызывается при его выходе из области видимости.

Семантика перемещения

Иногда может потребоваться передача права владения объектом, например при использовании `unique_ptr`. Нельзя скопировать `unique_ptr`, поскольку после уничтожения одной из копий `unique_ptr` оставшаяся часть `unique_ptr` будет содержать ссылку на удаленный объект. Вместо копирования объекта используется семантика перемещения в C++ для передачи владения от одного уникального указателя к другому, как показано в листинге 27.

Как и прежде, создается `unique_ptr<Foundation>` ❶. Он используется в течение некоторого времени, а затем нужно будет передать право владения объекту `Mutant`. Функция `move` сообщает компилятору, что вы хотите совершить передачу владения.

После создания the_mule ❷ время жизни Foundation связывается со временем жизни the_mule через переменную-член.

Листинг 27. Программа, перемещающая unique_ptr

```
#include <memory>

struct Foundation{
  const char* founder;
};

struct Mutant {
  // Конструктор устанавливает foundation соответствующим образом:
  Mutant(std::unique_ptr foundation)
    : foundation(std::move(foundation)) {}
  std::unique_ptr foundation;
};

int main() {
  std::unique_ptr second_foundation{ new Foundation{} }; ❶
  // ... используем second_foundation
  Mutant the_mule{ std::move(second_foundation) }; ❷
  // находится в 'перемещенном' состоянии
  // the_mule владеет классом Foundation
}
```

Расслабьтесь и получайте удовольствие

C++ является *основным* языком системного программирования. Большая часть ваших знаний по C будет применяться в C++, но вы также узнаете много новых концепций. Вы можете постепенно включать C++ в свои программы на C, используя Super C. По мере накопления знаний в некоторых более глубоких темах C++ вы обнаружите, что написание современного кода на C++ дает много преимуществ по сравнению с языком C. Вы сможете емко выражать идеи в коде, используя впечатляющую библиотеку stdlib, работать на более высоком уровне абстракции, использовать шаблоны для повышения производительности во время выполнения и повторного использования кода, а также опираться на жизненный цикл объекта C++ для управления ресурсами.

Я уверен, что инвестиции в изучение C++ принесут огромные дивиденды. А прочитав эту книгу, думаю, вы согласитесь со мной.