ЭФФЕКТИВНЫЙ C

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

РОБЕРТ С. СИКОРД

EFFECTIVE C

An Introduction to Professional C Programming

by Robert C. Seacord

no starch
press

San Francisco

ЭФФЕКТИВНЫЙ C

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

РОБЕРТ С. СИКОРД

ПИТЕР®

Санкт-Петербург • Москва • Минск

2022

ББК 32.973.2-018.1
УДК 004.43
С35

Сикорд Роберт С.

С35 Эффективный С. Профессиональное программирование. — СПб.: Питер, 2022. — 304 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1851-9

Мир работает на коде, написанном на С, но в большинстве учебных заведений программированию учат на Python или Java. Книга «Эффективный С. Профессиональное программирование» восполняет этот пробел и предлагает современный взгляд на С. Здесь рассмотрен С17, а также потенциальные возможности С2х. Издание неизбежно станет классикой, с его помощью вы научитесь писать профессиональные и надежные программы на С, которые лягут в основу устойчивых систем и решат реальные задачи.

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

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-1718501041 англ.

ISBN 978-5-4461-1851-9

Краткое содержание

Предисловие Паскаля Куока 14

Предисловие Олли Уайтхауса 15

Благодарности 16

Введение 17

Глава 1. Знакомство с C 24

Глава 2. Объекты, функции и типы 38

Глава 3. Арифметические типы 65

Глава 4. Выражения и операции 92

Глава 5. Управляющая логика 122

Глава 6. Динамически выделяемая память 142

Глава 7. Символы и строки 168

Глава 8. Ввод/вывод 204

Глава 9. Препроцессор (в соавторстве с Аароном Баллманом) 233

Глава 10. Структура программы (в соавторстве с Аароном Баллманом) 250

Глава 11. Отладка, тестирование и анализ 269

Список литературы 299

Об авторе 302

О соавторе 302

О научном редакторе 303

Оглавление

Предисловие Паскаля Куока 14

Предисловие Олли Уайтхауса 15

Благодарности 16

Введение 17
- Краткая история C 18
- Стандарт C 19
- Стандарт программирования CERT C 20
- Для кого эта книга 20
- Структура книги 21
- От издательства 23

Глава 1. Знакомство с C 24
- Разработка вашей первой программы на C 24
  - Компиляция и запуск вашей программы 25
  - Директивы препроцессора 26
  - Функция main 26
  - Проверка возвращаемого значения функции 28
  - Форматированный вывод 29
- Редакторы и интегрированные среды разработки 30
- Компиляторы 32
  - GNU Compiler Collection 32
  - Clang 33
  - Microsoft Visual Studio 33
- Переносимость 34
  - Поведение, определяемое реализацией 35
  - Неуточненное поведение 35
  - Неопределенное поведение 35
  - Поведение, зависящее от региональных параметров, и распространенные расширения 37
- Резюме 37

Глава 2. Объекты, функции и типы 38
Объекты, функции, типы и указатели 38
Объявление переменных 39
Перестановка значений местами (первая попытка) 40
Перестановка значений местами (вторая попытка) 42
Область видимости 44
Срок хранения 45
Выравнивание 47
Объектные типы 49
Логические типы 49
Символьные типы 49
Численные типы 50
Тип void 52
Функциональные типы 52
Производные типы 54
Указатели 54
Массивы 55
Структуры 57
Объединения 58
Теги 59
Квалификаторы типов 61
Квалификатор const 62
Квалификатор volatile 62
Квалификатор restrict 63
Упражнения 64
Резюме 64

Глава 3. Арифметические типы 65
Целые числа 65
Заполнение и точность 66
Заголовочный файл <limits.h> 66
Объявление целочисленных переменных 67
Беззнаковые целые 67
Знаковые целые 71
Целочисленные константы 76

Числа с плавающей запятой ... 77
Типы с плавающей запятой ... 78
Арифметические операции с плавающей запятой ... 80
Значения с плавающей запятой ... 80
Константы с плавающей запятой ... 82
Арифметическое преобразование ... 83
Ранг преобразования целочисленных типов ... 84
Повышение разрядности целочисленных значений ... 85
Обычные арифметические преобразования ... 86
Пример автоматического приведения типов ... 88
Безопасное приведение типов ... 89
Резюме ... 91

Глава 4. Выражения и операции ... 92
Простое присваивание ... 92
Оценивание ... 94
Вызов функции ... 95
Операции инкремента и декремента ... 96
Приоритет и ассоциативность операций ... 97
Порядок вычисления ... 100
Непоследовательные и неопределенно последовательные вычисления ... 101
Точки следования ... 102
Операция sizeof ... 103
Арифметические операции ... 104
Унарные операции + и – ... 104
Логическая операция отрицания ... 105
Мультипликативные операции ... 105
Аддитивные операции ... 107
Битовые операции ... 107
Операция дополнения ... 107
Операции сдвига ... 108
Битовое И ... 110
Битовое исключающее ИЛИ ... 111
Битовое включающее ИЛИ ... 112

Логические операции ... 112
Операции приведения типов ... 114
Условная операция ... 115
Операция _Alignof ... 116
Операции сравнения ... 117
Операции составного присваивания ... 118
Операция «запятая» ... 118
Арифметические операции с указателями ... 119
Резюме ... 121

Глава 5. Управляющая логика ... 122
Операторы-выражения ... 122
Составные операторы ... 123
Операторы выбора ... 124
Оператор if ... 124
Оператор switch ... 128
Операторы итерирования ... 131
Оператор while ... 131
Оператор do...while ... 133
Оператор for ... 134
Операторы перехода ... 136
Оператор goto ... 136
Оператор continue ... 138
Оператор break ... 139
Оператор return ... 140
Упражнения ... 141
Резюме ... 141

Глава 6. Динамически выделяемая память ... 142
Срок хранения ... 142
Куча и диспетчеры памяти ... 143
Когда следует использовать динамически выделяемую память ... 144
Функции для управления памятью ... 145
Функция malloc ... 145
Функция aligned_alloc ... 148

Функция calloc ... 149
Функция realloc ... 149
Функция reallocarray ... 152
Функция free ... 152
Состояния памяти ... 154
Структуры с массивами произвольной длины ... 155
Другие виды динамически выделяемой памяти ... 157
Функция alloca ... 157
Массивы переменной длины ... 159
Отладка проблем, связанных с выделением памяти ... 163
Библиотека dmalloc ... 163
Системы с повышенными требованиями к безопасности ... 166
Упражнения ... 166
Резюме ... 167

Глава 7. Символы и строки ... 168
Символы ... 168
ASCII ... 169
Unicode ... 169
Исходная и исполняемая кодировки ... 171
Типы данных ... 172
Символьные константы ... 175
Управляющие последовательности ... 176
Linux ... 177
Windows ... 178
Преобразование символов ... 180
Строки ... 184
Строковые литералы ... 186
Функции для работы со строками ... 188
Заголовочные файлы <string.h> и <wchar.h> ... 189
Интерфейсы с проверкой ограничений из приложения K ... 197
POSIX ... 200
Microsoft ... 202
Резюме ... 203

Глава 8. Ввод/вывод ... 204
Стандартные потоки ввода/вывода ... 204
Буферизация потоков ... 205
Потоки из стандартной библиотеки ... 206
Ориентация потоков ... 208
Текстовые и двоичные потоки ... 208
Открытие и создание файлов ... 209
Функция fopen ... 209
Функция open из стандарта POSIX ... 212
Закрытие файлов ... 214
Функция fclose ... 214
Функция close из стандарта POSIX ... 215
Чтение и запись символов и строк ... 216
Сброс потока на диск ... 218
Установка позиции в файле ... 219
Удаление и переименование файлов ... 222
Использование временных файлов ... 223
Чтение потоков форматированного текста ... 224
Чтение из двоичных потоков и запись в них ... 228
Резюме ... 232

Глава 9. Препроцессор (в соавторстве с Аароном Баллманом) ... 233
Процесс компиляции ... 233
Подключение файлов ... 235
Строки подключения с кавычками и угловыми скобками ... 236
Условная компиляция ... 237
Генерация ошибок ... 238
Использование стражей включения ... 239
Определение макросов ... 241
Макроподстановка ... 244
Макросы с обобщенными типами ... 246
Предустановленные макросы ... 248
Резюме ... 249

Глава 10. Структура программы (в соавторстве с Аароном Баллманом).... 250
Принципы компонентного представления 250
Связанность и слаженность 251
Повторное использование кода..... 252
Абстракция данных..... 253
Непрозрачные типы..... 254
Исполняемые файлы..... 256
Компоновка 258
Структурирование простой программы 260
Сборка кода..... 265
Резюме 268

Глава 11. Отладка, тестирование и анализ..... 269
Утверждения 269
Статические утверждения..... 269
Утверждения времени выполнения 272
Параметры и флаги компиляторов..... 274
GCC и Clang 275
Visual C++ 279
Отладка..... 281
Модульное тестирование 285
Статический анализ 289
Динамический анализ..... 292
AddressSanitizer..... 293
Упражнения..... 298
Резюме 298

Список литературы 299

Об авторе..... 302

О соавторе 302

О научном редакторе..... 303

Посвящается моим внучкам, Оливии и Изабелле,
а также другим молодым женщинам, которые станут
учеными и инженерами, когда вырастут.

Предисловие Паскаля Куока

Впервые о Роберте Сикорде я услышал в 2008 году. В то время он уже был широко известен среди программистов на C своей работой над *стандартом программирования CERT C* и приложением K к стандарту C. Однако на тот момент прошло лишь пара лет с тех пор, как я, молодой и глупый, присоединился к проекту Frama-C, призванному гарантировать отсутствие нежелательного поведения в программах на C. Однажды мой интерес привлекла заметка об уязвимостях от CERT, согласно которой в кое-каких компиляторах C (в частности, в GCC) были убраны определенные проверки переполнения операций с плавающей запятой. Это решение имело причину: проверки были слишком примитивными и в случае переполнения приводили к неопределенному поведению.

Кроме того, компиляторы не были обязаны сообщать программисту о том, в чем состояла его ошибка, даже при максимальном уровне предупреждений. Неопределенное поведение в C может иметь серьезные последствия. Я решил исправить эту конкретную проблему. Одним из авторов той заметки был Роберт.

Эта книга научит вас программировать на C в современных условиях. Она поможет выработать хорошие привычки, которые не позволят вам сознательно или по небрежности использовать нежелательное поведение. Читатель должен понимать, что в объемных программах на языке C такое поведение может быть вызвано произвольным вводом, поэтому устранения обычных ошибок программистов может быть недостаточно!

В этой книге делается беспрецедентный упор на аспекты программирования на C, связанные с безопасностью. Я рекомендую к использованию все представленные в ней инструменты — они помогут вам избежать неопределенного поведения в ваших программах.

Паскаль Куок, главный научный сотрудник, TrustInSoft

Предисловие Олли Уайтхауса

Начав заниматься кибербезопасностью больше 25 лет назад, я изучал свое ремесло в основном путем поиска и использования небезопасного кода для работы с памятью в программах на C. Даже тогда этому классу уязвимостей было больше 20 лет. Работая в BlackBerry и перебирая горы отчетов о разборе кода, я мог воочию убедиться в том, насколько опасным может быть язык C для непосвященных. Теперь, уже в качестве технического директора транснациональной компании, предоставляющей профессиональные услуги по обеспечению кибербезопасности, я каждый день вижу, чем чреват плохо написанный код на C для нашего общества в цифровую эпоху.

Написание безопасного и профессионального кода на C до сих пор вызывает множество трудностей. Это может свести (и регулярно сводит) на нет многочисленные инновации в средствах защиты компиляторов и операционных систем. И хотя мы наблюдаем существенный прогресс в других современных языках, спрос на C по-прежнему растет, особенно в IoT и прочих средах с крайне ограниченными ресурсами.

Мнение Роберта о том, как профессионально и безопасно программировать на C, пользуется большим уважением. Уже больше десяти лет я советую его материал как клиентам, так и разработчикам внутри компании. Никто лучше него не научит вас писать код на C профессионально и среди прочего безопасно.

На сегодня профессиональный код на C должен быть производительным, безопасным и защищенным. Те, кто в состоянии выполнять эти требования, смогут приносить пользу обществу, не создавая дополнительных технических проблем, которые придется решать в будущем.

Эта книга поможет читателям с минимальным опытом использования C (или вообще без такового) быстро обрести знания и навыки, которые позволят им стать профессиональными программистами на этом языке и сформируют прочный фундамент для разработки высокопроизводительных, защищенных и безопасных систем.

Олли Уайтхаус, технический директор, NCC Group

Благодарности

Хочу отметить всех тех, кто помогал в создании этой книги. Начну с Билла Поллока из No Starch Press, который неустанно убеждал меня написать книгу о C.

Благодарю Олли Уайтхауса и Паскаля Куока за отличные предисловия.

Аарон Баллман был прекрасным партнером в этом начинании. Помимо двух написанных им глав, он вычитывал весь текст (часто по несколько раз) и помогал мне решать как сложные, так и тривиальные проблемы.

Дуглас Гвин, заслуженный сотрудник научно-исследовательской лаборатории сухопутных войск США и почетный член комитета во главе стандарта C, помог вычитать все главы. Когда мои писательские навыки не соответствовали его ожиданиям, он подталкивал меня в правильном направлении.

Официальным научным редактором этой книги был мой хороший друг, Мартин Себор, поэтому за любые неточности, которые вы найдете, винить нужно именно его.

Помимо Аарона, Дага и Мартина отдельные главы вычитывали и другие уважаемые члены комитетов во главе стандартов C и C++, включая Джима Томаса, Томаса Кёппе, Нилла Дугласа, Тома Хонерманна и Жана Хейда Менеида. В число научных редакторов вошли мои коллеги из NCC Group: Ник Данн, Джонатан Линдси, Томаш Крамковски, Алекс Донисторп, Джошуа Доу, Каталин Висинеску, Аарон Адамс и Саймон Харраги. Техническими редакторами, не имеющими отношения к этим организациям, были Дэвид Леблан, Николас Уинтер, Джон Макфарлейн и Скотт Алоизио.

Кроме того, я хотел бы поблагодарить следующих профессионалов из No Starch, которые обеспечили выпуск качественного продукта: Элизабет Чедвик, Фрэнсис Со, Зака Лебовски, Энни Чой, Барбару Йен, Катрину Тейлор, Натали Глисон, Дерека Йи, Лорен Чун, Джину Редман, Шерон Уилки, Эмили Баттаглиа и Дапиндера Досанжа.

Введение

Язык C был разработан для системного программирования еще в 1970-х, но до сих пор остается невероятно популярным. Системные языки рассчитаны на производительность и простоту доступа к внутреннему аппаратному обеспечению, но при этом обладают высокоуровневыми возможностями. Существуют более современные языки, но их компиляторы и библиотеки обычно написаны на C. Как однажды сказал Карл Саган, «если вы хотите приготовить яблочный пирог с нуля, то вам сначала нужно создать вселенную».

Создателям C не пришлось заходить настолько далеко; они спроектировали этот язык для работы с разнообразным вычислительным оборудованием и архитектурами, которые, в свою очередь, были ограничены законами физики и математики. Среда выполнения C находится непосредственно над аппаратным обеспечением, что делает ее более чувствительной к появлению оборудования с новыми возможностями, такими как векторные инструкции, эффективность которых в языках более высокого уровня обычно зависит от C.

Согласно рейтингу TIOBE, с 2001 года C остается либо самым популярным, либо вторым по популярности языком программирования[1]. В 2019 году C стал языком года по версии TIOBE. Своей популярностью он, скорее всего, обязан нескольким основополагающим принципам, известным как *дух C*.

- Доверие к программисту. В целом язык C подразумевает, что вы знаете, что делаете, и позволяет вам делать это. Иногда такие действия имеют плохие последствия (например, когда вы не знаете, что делаете).

[1] Рейтинг TIOBE оценивает популярность языков программирования и находится по адресу https://www.tiobe.com/tiobe-index/. При его составлении учитываются количество квалифицированных программистов, учебных курсов и сторонних поставщиков для каждого языка. Этот рейтинг может пригодиться, если вы выбираете язык для изучения или разработки новой программной системы.

- Не следует мешать программисту делать то, что нужно. Язык C предназначен для системного программирования, поэтому должен уметь справляться с разнообразными низкоуровневыми задачами.
- Язык должен быть компактным и простым. Язык C достаточно сильно связан с оборудованием и является легковесным.
- Для выполнения любой операции должен быть лишь один способ.
- Язык должен быть быстрым, даже если его переносимость не гарантируется. Главный приоритет — возможность написания оптимально эффективного кода. А сделать его переносимым, безопасным и защищенным должны вы, программист.

Краткая история C

Язык программирования C был создан в 1972 году Деннисом Ритчи и Кеном Томпсоном в лабораториях Bell Labs. Позже Деннис Ритчи вместе с Брайаном Керниганом написал книгу *The C Programming Language*[1] (K&R, 1988). В 1983 году Американский национальный институт стандартов (American National Standards Institute, ANSI) сформировал комитет X3J11 для разработки стандартной спецификации C, а в 1989 году стандарт C был ратифицирован как ANSI X3.159-1989, «Язык программирования C». Эта версия языка называется *ANSI C* или *C89*.

В 1990 году стандарт *ANSI C* был принят (без изменений) совместным комитетом Международной организации по стандартизации (International Organization for Standardization, ISO) и Международной электротехнической комиссией (International Electrotechnical Commission, IEC) и опубликован в качестве первой редакции стандарта C, C90 (ISO/IEC 9899:1990). Вторая редакция, C99, была опубликована в 1999 году (ISO/IEC 9899:1999), а третья, C11, — в 2011-м (ISO/IEC 9899:2011). Последняя (на момент написания этих строк), четвертая версия стандарта C вышла в 2018 году под названием C17 (ISO/IEC 9899:2018). В настоящий момент ISO/IEC работают над новейшей редакцией языка C, известной как C2x. Согласно опросу, проведенному компанией JetBrains в 2018 году, 52 % программистов на C используют C99, 36 % — C11, а 23 % работают с языком C для встраиваемых систем[2].

---

[1] *Керниган Б., Ритчи Д.* Язык программирования C. — М.: Вильямс, 2015.

[2] Подробности исследования ищите на сайте JetBrains: https://www.jetbrains.com/lp/devecosystem-2019/c/.

Стандарт C

Стандарт C (ISO/IEC 9899:2018) описывает язык и является главным его авторитетом. Он может казаться неясным или даже непостижимым, но если вы хотите писать переносимый, безопасный и защищенный код, то вам необходимо понимать этот стандарт. Он предоставляет реализациям существенную степень свободы, позволяя им оставаться оптимально эффективными на разных аппаратных платформах. *Реализация* в терминологии стандарта C обозначает компилятор и имеет следующее определение:

> «Определенный набор программного обеспечения, работающий в конкретной среде трансляции с определенными управляющими параметрами и отвечающий за трансляцию программ для конкретной среды выполнения с поддержкой работы функций».

Из этого определения следует, что каждый компилятор с неким набором флагов командной строки и стандартной библиотекой C считается отдельной реализацией и что *поведение* разных реализаций может существенно различаться. Это видно на примере компилятора GNU GCC, который использует флаг `-std=` для определения стандарта языка; для данного параметра можно указывать такие значения, как `c89`, `c90`, `c99`, `c11`, `c17`, `c18` и `c2x`. Значение по умолчанию зависит от версии компилятора. Если диалект C не был указан, то GCC 10 выбирает `-std=gnu17`, предоставляя тем самым расширения языка C. Если вам нужна переносимость, указывайте используемый вами стандарт. Для доступа к новым возможностям языка выбирайте последнюю спецификацию. Так, в 2019 году хорошим выбором был `-std=c17`.

Поскольку реализации могут вести себя по-разному и некоторые варианты их поведения являются неопределенными, то для того, чтобы понимать язык C, недостаточно писать простые тестовые программы[1]. При компиляции программы с помощью разных реализаций и на разных платформах ее поведение может меняться; расхождения могут возникать даже при использовании разных наборов флагов или стандартных библиотек C в рамках одной и той же реализации. Поведение кода может зависеть даже от разных *версий* компилятора. Стандарт C — единственный документ, описывающий, какие аспекты поведения гарантируются во всех реализациях, а какие могут варьироваться. Это в основном имеет последствия для разработки переносимых программ, но может сказаться и на защищенности и безопасности вашего кода.

[1] Для этого есть замечательный инструмент Compiler Explorer: https://godbolt.org/.

Стандарт программирования CERT C

Руководя командой, отвечающей за безопасность кода в Институте программной инженерии при Университете Карнеги — Меллона, я написал справочное пособие *The CERT® C Coding Standard, Second Edition: 98 Rules for Developing Safe, Reliable, and Secure Systems* (Сикорд, 2014). В нем представлены примеры распространенных ошибок программирования на C и рекомендации по их исправлению. В данной книге некоторые из этих рекомендаций упоминаются в качестве источника подробной информации о тех или иных аспектах программирования на языке C.

Для кого эта книга

Эта книга представляет собой вводный материал по языку C. Она написана так, чтобы быть понятной любому, кто хочет научиться программировать на данном языке, но обходится без излишних упрощений, которыми грешат многие другие вводные книги и учебные курсы. Слишком элементарный материал научит вас писать код, который будет компилироваться и работать, но при этом может содержать ошибки. Разработчики, обучающиеся программированию на C с помощью таких пособий, обычно пишут некачественный, дефектный, небезопасный код, который в конечном счете придется переписывать (и зачастую довольно скоро). Надеюсь, их более опытные коллеги рано или поздно помогут им выбросить из головы эти вредные заблуждения о программировании на C и научат разрабатывать профессиональный и качественный код. С другой стороны, эта книга быстро научит вас писать правильный, переносимый код профессионального качества, заложит фундамент для создания систем с повышенными требованиями к защищенности и безопасности и, возможно, поможет вам усвоить несколько вещей, о которых не знают даже более опытные разработчики.

Книга содержит краткое введение в основы программирования на языке C, и после ее прочтения вы сможете писать программы, решать задачи и создавать рабочие системы. Здесь вы найдете понятные идиоматические примеры кода.

Вы познакомитесь с ключевыми концепциями программирования на C и научитесь писать высококачественный код, выполняя упражнения для каждой рассматриваемой темы. Вы узнаете, какие методики рекоменду-

ются для разработки правильного и безопасного кода на C. Обновления и дополнительный материал можно найти на веб-странице этой книги https://www.nostarch.com/effective_c/ и на сайте http://www.robertseacord.com/. Если после прочтения вам захочется узнать больше о безопасном программировании на C, C++ или других языках, то, пожалуйста, ознакомьтесь с учебными курсами, которые предлагает NCC Group, пройдя по адресу https://www.nccgroup.trust/us/our-services/cyber-security/security-training/secure-coding/.

Структура книги

Данная книга начинается с вводной главы, которая охватывает ровно столько материала, сколько нужно, чтобы начать программировать. Затем мы сделаем шаг назад и исследуем главные составные элементы языка. В двух завершающих главах вы научитесь создавать из этих элементов настоящие системы и узнаете, как отлаживать, тестировать и анализировать написанный вами код. Главы выглядят следующим образом.

- В *главе 1 «Знакомство с C»* вы напишете простую программу на C, чтобы научиться использовать функцию `main`. Мы также рассмотрим несколько разных редакторов и компиляторов.
- В *главе 2 «Объекты, функции и типы»* изложены основы языка, такие как объявление переменных и функций. Вы также познакомитесь с принципами использования базовых типов.
- В *главе 3 «Арифметические типы»* рассказано о двух видах арифметических типов данных: целочисленных и с плавающей запятой.
- В *главе 4 «Выражения и операции»* вы узнаете, что такое операции[1], и научитесь писать простые выражения для выполнения операций с объектами различных типов.
- В *главе 5 «Управляющая логика»* вы узнаете, как управлять порядком вычисления отдельных операторов. Для начала мы пройдемся по операторам-выражениям и составным операторам, которые описывают,

[1] Английское слово operator, соответствующее термину «операция», иногда ошибочно переводят как «оператор». На самом деле (по историческим причинам) русский термин «оператор» соответствует английскому statement. Разговаривая с коллегами, скорее всего, вы будете использовать термин «оператор» как аналог англоязычного operator. — *Примеч. пер.*

какую работу нужно проделать. Затем рассмотрим три вида операторов, которые определяют, какие блоки кода выполняются и в каком порядке это происходит: операторы выбора, итерирования и перехода.

- В *главе 6 «Динамически выделяемая память»* вы узнаете, как память динамически выделяется в куче во время работы программы. Она нужна в ситуациях, когда до запуска кода неизвестно, сколько памяти ему понадобится.
- В *главе 7 «Символы и строки»* вы изучите различные кодировки, включая ASCII и Unicode, которые можно использовать для составления строк. Вы научитесь применять устаревшие функции из стандартной библиотеки C, интерфейсы с проверкой ограничений, а также API POSIX и Windows для представления и изменения строк.
- В *главе 8 «Ввод/вывод»* вы научитесь выполнять операции ввода/вывода для чтения и записи данных в терминалы и файловые системы. Ввод/вывод охватывает все пути, которыми информация попадает в программу и покидает ее. Без этого ваш код был бы бесполезным. Мы рассмотрим методики на основе стандартных потоков C и файловых дескрипторов POSIX.
- В *главе 9 «Препроцессор»* вы научитесь использовать препроцессор для подключения файлов, определения объектных и функциональных макросов, а также научитесь выполнять условную компиляцию кода в зависимости от свойств той или иной реализации.
- В *главе 10 «Структура программы»* вы научитесь разбивать свои программы на разные единицы трансляции, состоящие как из исходных, так и заголовочных файлов. Кроме того, вы узнаете, как скомпоновать несколько объектных файлов в единое целое, чтобы создать библиотеку или исполняемую программу.
- В *главе 11 «Отладка, тестирование и анализ»* описываются инструменты и методики для создания корректных программ, включая утверждение времени компиляции и выполнения, отладку, тестирование, статический и динамический анализ. Мы также обсудим, какие флаги компиляции рекомендуется использовать на разных этапах процесса разработки программного обеспечения.

Вам предстоит путешествие, по завершении которого вы станете новоиспеченным, но профессиональным программистом на C.

От издательства

Ваши замечания, предложения, вопросы отправляйте по электронному 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.

1 Знакомство с C

В этой главе вы разработаете свою первую программу на C, которая будет выводить традиционное сообщение Hello, world!. Я расскажу вам о ее различных аспектах и покажу, как ее компилировать и запускать. Затем мы обсудим несколько доступных редакторов и компиляторов, а также распространенные проблемы с переносимостью, с которыми вы очень скоро столкнетесь, программируя на C.

Разработка вашей первой программы на C

Изучение программирования на C лучше всего начать с написания программ, и традиционной программой, с которой принято начинать, является Hello, world!.

Для написания данной программы нам понадобится текстовый редактор или *интегрированная среда разработки* (integrated development environment, IDE). Выбрать есть из чего, но пока можете открыть свой любимый редактор, а другие варианты мы рассмотрим позже в этой главе.

Введите в своем текстовом редакторе программу из листинга 1.1.

Листинг 1.1. Программа hello.c

```
  #include <stdio.h>
  #include <stdlib.h>
❶ int main(void) {
  ❷ puts("Hello, world!");
  ❸ return EXIT_SUCCESS;
❹ }
```

Чуть позже мы пройдемся по каждой строчке этого кода. Но пока сохраните его в файл `hello.c`. Расширение `.c` означает, что данный файл содержит исходный код на языке C.

ПРИМЕЧАНИЕ

Если вы приобрели электронную версию книги, то скопируйте код программы в редактор. По возможности используйте функции копирования и вставки, поскольку это может избавить вас от опечаток при ручном вводе.

Компиляция и запуск вашей программы

Дальше нам нужно скомпилировать и запустить эту программу. Данный процесс состоит из двух отдельных этапов. Вы можете выбрать один из множества компиляторов C, и от этого выбора будет зависеть команда для компиляции программы. В Linux и других Unix-подобных операционных системах можно воспользоваться системным компилятором с помощью команды `cc`. Чтобы скомпилировать свою программу, введите в командной строке **`cc`** и укажите имя соответствующего файла:

```
% cc hello.c
```

ПРИМЕЧАНИЕ

Эти команды относятся к Linux (и другим Unix-подобным операционным системам). Другие компиляторы в других операционных системах нужно запускать по-другому. Сверьтесь с документацией вашего конкретного компилятора.

Если вы правильно ввели программу, то команда компиляции создаст в каталоге с вашим исходным кодом новый файл `a.out`. Просмотрите содержимое своего каталога с помощью команды `ls`. Вы должны увидеть следующее:

```
% ls
a.out hello.c
```

Файл `a.out` — это исполняемая программа, которую можно запустить в командной строке:

```
% ./a.out
Hello, world!
```

Если все было сделано правильно, то программа должна вывести в окно терминала `Hello, world!`. В противном случае сравните исходный текст в листинге 1.1 со своим кодом и убедитесь, что они совпадают.

У команды `cc` есть множество флагов и параметров компиляции. Флаг `-o file`, к примеру, позволяет назначить исполняемому файлу более запоминающееся имя вместо `a.out`. Следующая команда компиляции называет исполняемый файл `hello`:

```
% cc -o hello hello.c
% ./hello
Hello, world!
```

Теперь проанализируем программу `hello.c` строчка за строчкой.

Директивы препроцессора

В первых двух строчках программы `hello.c` используется директива препроцессора `#include`, которая ведет себя так, будто вы подставили вместо нее содержимое указанного файла. Мы подключаем заголовочные файлы `<stdio.h>` и `<stdlib.h>`, чтобы получить доступ к объявленным в них функциям, которые затем сможем вызывать в своем коде. Функция `puts` объявлена в `<stdio.h>`, а макрос `EXIT_SUCCESS` — в `<stdlib.h>`. Как можно догадаться по названиям этих файлов, `<stdio.h>` содержит объявления функций ввода/вывода (I/O), стандартных для C, а в `<stdlib.h>` находятся объявления служебных функций общего назначения. Вам нужно подключить объявления всех библиотечных функций, которые используются в вашей программе.

Функция main

Главная часть программы, показанной в листинге 1.1, начинается со строчки ❶:

```
int main(void) {
```

Данная строчка определяет функцию `main`, которая вызывается при запуске программы. Это главная точка входа программы, которая выполняется в серверной среде, когда программа запускается из командной строки или другой программы. Язык C имеет две потенциальные среды выполнения: *минимальную (freestanding)* и *полноценную (hosted)*. Минимальная может существовать вне ОС и обычно используется в программировании

встраиваемых систем. Такие среды предоставляют минимальный набор библиотечных функций, а название и тип точки входа, которая вызывается при запуске программы, зависит от реализации. Эта книга в основном ориентирована на полноценные среды.

Мы определили функцию `main` так, чтобы она возвращала значение типа `int`, и указали `void` в скобках; это значит, она не принимает никаких аргументов. Тип `int` — это знаковый целочисленный тип, который можно использовать для представления как положительных, так и отрицательных целых значений (а также 0). По аналогии с другими процедурными языками программы на C состоят из процедур (называемых *функциями*), которые могут принимать аргументы и возвращать значения. Каждую функцию можно вызывать столько раз, сколько потребуется. В данном случае значение, возвращаемое функцией `main`, говорит о том, завершилась ли программа успешно. Работа, выполняемая этой функцией ❷, состоит в выводе строки `Hello, world!`:

```
puts("Hello, world!");
```

Функция `puts` входит в стандартную библиотеку C и записывает строковый аргумент в поток `stdout`, который обычно представляет консоль или окно терминала, и добавляет к выводу символ перевода строки. `"Hello, world!"` — это строковый литерал, который ведет себя как строка, доступная только для чтения. Вызов этой функции выводит `Hello, world!` в терминал.

Вы хотите, чтобы программа закрылась, когда она выполнит работу. Это можно выполнить с помощью оператора `return` ❸ внутри функции `main`, чтобы вернуть целочисленное значение в серверную среду или вызывающий скрипт:

```
return EXIT_SUCCESS;
```

`EXIT_SUCCESS` — это объектоподобный макрос, который обычно разворачивается в `0` и, как правило, имеет следующее определение:

```
#define EXIT_SUCCESS 0
```

Вместо каждого упоминания `EXIT_SUCCESS` подставляется `0`, который затем возвращается внешней среде исполнения из вызова функции `main`. Затем скрипт, вызывавший программу, может проверить ее состояние и определить, был ли ее вызов успешным. Возвращение из исходного вызова функции `main` эквивалентно вызову стандартной библиотечной функции `exit` со значением, возвращенным функцией `main` в качестве аргумента.

Заключительная строчка этой программы ❹ содержит закрывающую фигурную скобку, которая завершает блок кода, начатый нами с объявления функции `main`:

```
int main(void) {
  // ---snip---
}
```

Открывающую скобку можно разместить в одной строчке с объявлением или отдельно, как показано ниже:

```
int main(void)
{
  // ---snip---
}
```

Это сугубо стилистическое решение, поскольку пробельные символы (включая перевод строки) в целом не влияют на синтаксис. В данной книге я обычно указываю открывающую фигурную скобку в одной строчке с объявлением функции, так как это более компактный стиль.

Проверка возвращаемого значения функции

Функции зачастую возвращают значения, которые являются результатом вычислений или указывают на то, успешно ли выполнена задача. Например, функция `puts`, которую мы использовали в нашей программе Hello, world!, принимает строку, которую нужно вывести, и возвращает значение типа `int`. Если произошла ошибка, то это значение равно макросу `EOF` (отрицательному целому числу); в противном случае возвращается неотрицательное целое значение.

И хотя вероятность того, что функция `puts` в нашей простой программе завершится неудачно и вернет `EOF`, очень низкая, это может произойти. Поскольку вызов `puts` может провалиться и вернуть `EOF`, это означает, что ваша первая программа на C имеет дефект или по крайней мере может быть улучшена следующим образом:

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
  if (puts("Hello, world!") == EOF) {
    return EXIT_FAILURE;
    // здесь код никогда не выполняется
  }
```

```
  return EXIT_SUCCESS;
  // здесь код никогда не выполняется
}
```

Эта переработанная версия программы Hello, world! проверяет, возвращает ли вызов `puts` значение `EOF`, сигнализирующее об ошибке записи. Если мы получаем `EOF`, то программа возвращает значение макроса `EXIT_FAILURE` (который соответствует ненулевому числу). Если нет, то функция завершается успешно, а программа возвращает макрос `EXIT_SUCCESS` (который должен быть равен 0). Скрипт, вызывающий программу, может проверить ее состояние, чтобы определить, было ли ее выполнение успешным. Код, идущий за оператором `return`, является *мертвым* («недостижимым») и не выполняется никогда. Об этом сигнализируют однострочные комментарии в переработанной версии программы. Компилятор игнорирует все, что находится после `//`.

Форматированный вывод

Функция `puts` позволяет простым и удобным способом вывести строки в `stdout`. Но рано или поздно вам нужно будет вывести отформатированный текст с помощью функции `printf` — например, чтобы отобразить нестроковые аргументы. Функция `printf` принимает строку, описывающую формат вывода, и произвольное количество аргументов, представляющих собой значения, которые вы хотите вывести. Например, если вы хотите отобразить с помощью функции `printf` строку `Hello, world!`, то это можно сделать так:

```
printf("%s\n", "Hello, world!");
```

Первый аргумент — это строка форматирования `"%s\n"`. Элемент `%s` — спецификация преобразования, которая заставляет функцию `printf` прочитать второй аргумент (строковой литерал) и вывести его в `stdout`. Элемент `\n` — это алфавитная управляющая последовательность, предназначенная для представления невидимых символов; она сообщает функции о том, что после текста нужно добавить перевод строки. Без этой последовательности следующие символы (скорее всего, приглашение командной строки) отображались бы в той же строчке. Этот вызов имеет следующий вывод:

```
Hello, world!
```

Следите за тем, чтобы данные, предоставленные пользователем, не стали первым аргументом функции `printf`, поскольку это чревато уязвимостью безопасности на основе отформатированного вывода (Сикорд, 2013).

Как уже было показано ранее, самый простой способ вывести строки — использовать функцию `puts`. Если в переработанной версии программы Hello, world! заменить ее функцией `printf`, то ваш код перестанет работать, поскольку `printf` возвращает состояние не так, как `puts`. В случае успеха функция `printf` возвращает количество выведенных символов, а при возникновении ошибки — отрицательное значение. Попробуйте в качестве упражнения сделать так, чтобы программа Hello, world! использовала `printf`.

Редакторы и интегрированные среды разработки

Для разработки программ на C можно использовать всевозможные редакторы и IDE. На рис. 1.1 перечислены самые популярные из них, согласно исследованию JetBrains за 2018 год.

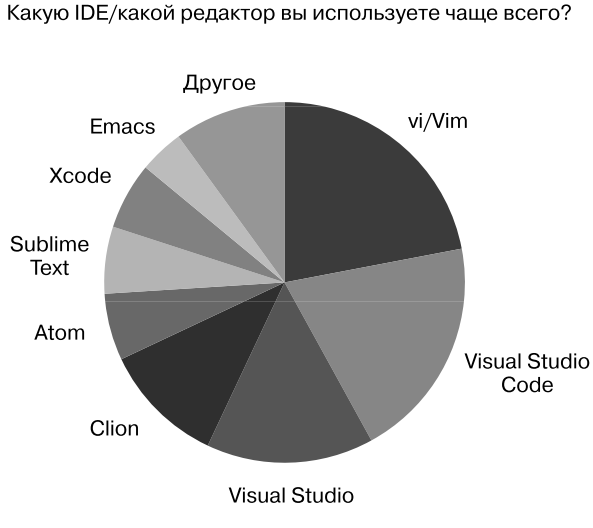

Рис. 1.1. Использование IDE/редакторов

То, какие именно инструменты вам доступны, зависит от используемой вами системы. Данная книга ориентирована на Linux, Windows и macOS, поскольку это самые распространенные платформы для разработки.

Для Microsoft Windows очевидным выбором является Visual Studio IDE (https://visualstudio.microsoft.com/). Она предлагается в трех вариантах: Community, Professional и Enterprise. Преимущество версии Community состоит в ее бесплатности, а другие варианты имеют дополнительные, хоть и платные возможности. В этой книге вам будет достаточно версии Community.

В Linux все не так очевидно. Вы можете выбрать Vim, Emacs, Visual Studio Code или Eclipse. Vim — любимый текстовый редактор многих программистов и опытных пользователей. Он основан на проекте `vi`, написанном Биллом Джоем для одной из разновидностей Unix. Vim унаследовал от `vi` сочетания клавиш, но при этом предлагает дополнительные функции и расширяемость, которой недостает оригиналу. При желании для Vim можно установить подключаемые модули, такие как YouCompleteMe (https://github.com/Valloric/YouCompleteMe/) или deoplete (https://github.com/Shougo/deoplete.nvim/), которые предоставляют встроенное семантическое дополнение кода для программирования на C.

GNU Emacs — расширяемый, гибкий и бесплатный текстовый редактор. По своей сути это интерпретатор для Emacs Lisp, диалекта языка программирования Lisp с поддержкой расширений для редактирования текста — хотя для меня это никогда не было проблемой. Признаюсь: почти весь код на C, который я когда-либо разработал, был написан в Emacs.

Visual Studio Code (VS Code) — это удобный в использовании редактор кода с поддержкой таких аспектов разработки, как отладка, запуск заданий и управление версиями (подробности — в главе 11). Он предоставляет именно те инструменты, которые нужны программисту для быстрого цикла «программирование — сборка — отладка». VS Code работает на macOS, Linux и Windows и бесплатен для личного и коммерческого использования. Инструкции по установке доступны для Linux и других платформ[1]; в Windows вы, скорее всего, остановите свой выбор на Microsoft Visual Studio. На рис. 1.2 показан редактор Visual Studio Code в Ubuntu, с помощью которого написана программа из листинга 1.1. Как видно в консоли отладки, программа, как и ожидалось, вышла с кодом состояния 0.

[1] Инструкции по установке в Linux находятся на странице Visual Studio Code on Linux по адресу https://code.visualstudio.com/docs/setup/linux/.

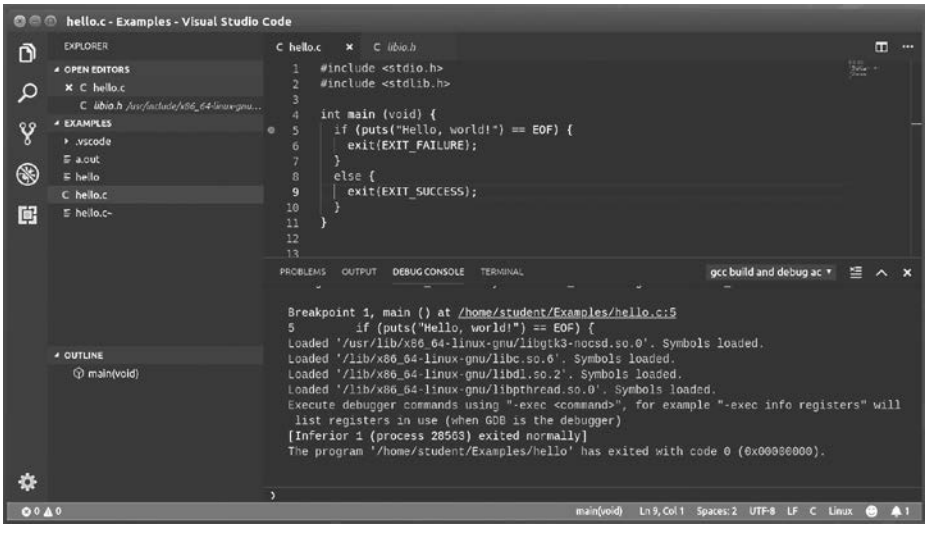

Рис. 1.2. Visual Studio Code в Ubuntu

Компиляторы

Компиляторов C много, поэтому я не стану обсуждать их все. Каждый из них реализует определенные версии стандарта C. Многие компиляторы для встраиваемых систем поддерживают только C89/C90. Популярные компиляторы для Linux и Windows пытаются поддерживать современные стандарты, включая C2x.

GNU Compiler Collection

GCC (GNU Compiler Collection — набор компиляторов GNU) поддерживает C, C++, Objective-C и другие языки (https://gcc.gnu.org/). Разработка GCC проводится в соответствии с четким планом и под надзором руководящего комитета.

GCC — стандартный компилятор в системах Linux, хотя у него также есть версии для Microsoft Windows, macOS и других платформ. Установить GCC в Linux довольно просто. Например, ниже показана команда для установки GCC 8 в Ubuntu:

```
% sudo apt-get install gcc-8
```

Следующая команда позволяет проверить версию GCC, которая у вас установлена:

```
% gcc --version
gcc (Ubuntu 8.3.0-6ubuntu1~18.04) 8.3.0
This is free software; see the source for copying conditions. There is NO
Warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Если вы собираетесь разрабатывать программное обеспечение для Red Hat Enterprise Linux, то идеальной системой для этого будет Fedora. Ниже показано, как установить GCC в данной системе:

```
% sudo dnf install gcc
```

Clang

Еще один популярный компилятор — *Clang* (https://clang.llvm.org/). Установить Clang в Linux не составляет труда. Например, вот как это делается в Ubuntu:

```
% sudo apt-get install clang
```

Проверить, какая версия Clang у вас установлена, можно с помощью следующей команды:

```
% clang --version
clang version 6.0.0-1ubuntu2 (tags/RELEASE_600/final)
Target: x86_64-pc-linux-gnu
Thread model: posix
InstalledDir: /usr/bin
```

Microsoft Visual Studio

Microsoft Visual Studio — самая популярная среда разработки для Windows, которая включает в себя как IDE, так и компилятор. На момент написания этой книги самой последней версией является Visual Studio 2019 (https://visualstudio.microsoft.com/downloads/). Она поставляется вместе с пакетом Visual C++ 2019, в который входят компиляторы для C и C++.

Параметры для Visual Studio можно указывать на экранах Project Property. Откройте вкладку Advanced в разделе C/C++ и убедитесь в том, что ваш код компилируется в режиме C; для этого должен быть выбран пункт Compile as C Code (/TC), а не Compile as C++ Code (/TP). Если файл имеет расширение `.c`, то по умолчанию компилируется с параметром `/TC`. Если же используется

расширение `.cpp`, `.cxx` или одно из нескольких других, то компиляция происходит с `/TP`.

Переносимость

Все реализации компилятора C имеют некие особенности. Они постоянно развиваются, поэтому, к примеру, такой компилятор, как GCC, может иметь полноценную поддержку C17, а работы над поддержкой C2x еще ведутся; в этом случае одни возможности C2x будут поддерживаться, а другие — нет. Следовательно, компиляторы не поддерживают весь спектр спецификаций стандарта C (в том числе и промежуточных). В целом реализации C развиваются медленно, и многие компиляторы существенно отстают от стандарта языка.

Программы, написанные на C, можно считать *строго соответствующими* спецификации, если они используют только те возможности языка и библиотеки, которые предусмотрены стандартом. Такие программы должны быть максимально переносимыми. Но в реальности из-за широкого разнообразия реализаций с разным поведением никакие программы на C не могут (и никогда не смогут) похвастаться строгим соблюдением спецификации, и это, вероятно, нормально. Вместо этого можно создавать программы, *частично соответствующие* стандарту C, которые могут зависеть от непереносимых возможностей языка и библиотеки.

Зачастую код пишут для какой-то одной или нескольких эталонных реализаций, в зависимости от того, на каких платформах его планируется использовать. Стандарт C гарантирует, что эти реализации если и отличаются, то не слишком сильно, и позволяет вам ориентироваться сразу на несколько из них, не прибегая к необходимости изучать новый язык в каждом отдельном случае.

В дополнении J к стандарту C перечислено пять видов проблем переносимости:

- поведение, определяемое реализацией;
- неуточненное поведение;
- неопределенное поведение;
- поведение, зависящее от региональных параметров;
- распространенные расширения.

В процессе изучения языка C вам будут встречаться примеры всех пяти видов поведения, поэтому вы должны понимать, что именно они собой представляют.

Поведение, определяемое реализацией

Поведение, определяемое реализацией (implementation-defined behavior), не оговаривается стандартом C и может иметь разные результаты в разных реализациях. При этом в рамках отдельно взятой реализации это поведение является предсказуемым и задокументированным. Примером может служить количество битов в байте.

Поведение, определяемое реализацией, в большинстве случаев не создает проблем, но может приводить к появлению дефектов при переносе кода на другую реализацию языка. Старайтесь не писать код, поведение которого может варьироваться в зависимости от реализации C, которую вы используете для его компиляции. Полный список всех примеров поведения, определяемого реализацией, приводится в дополнении J.3 к стандарту C. Вы можете документировать зависимость своего кода от такого поведения с помощью объявления `static_assert`, как будет показано в главе 11. На страницах этой книги я отмечаю, когда поведение кода зависит от реализации.

Неуточненное поведение

Неуточненное поведение (unspecified behavior) наблюдается, когда стандарт предусматривает несколько вариантов выполнения, но не уточняет, какой из них должен использоваться в том или ином случае. При каждом выполнении заданное выражение может иметь разные результаты или возвращать разные значения. Примером неуточненного поведения служит размещение параметров функции в памяти, которое может меняться при каждом вызове этой функции в рамках одной и той же программы. Старайтесь не писать код, зависящий от неуточненного поведения, примеры которого перечислены в дополнении J.1 к стандарту C.

Неопределенное поведение

Неопределенным (undefined) является поведение, которое не определено в стандарте C, или, чтобы не ходить по кругу, «поведение при использовании непереносимых или ошибочных программных конструкций либо некорректных данных, к которому стандарт не предъявляет никаких

требований». К неопределенному поведению относят переполнение знакового целого и разыменование некорректного указателя. Код, содержащий неопределенное поведение, как правило, является ошибочным, но с нюансами. В стандарте неопределенное поведение описывается так:

- когда требование вида «должно» или «не должно» нарушается и, по всей видимости, выходит за рамки ограничения, поведение не определено;
- когда поведение явно описано как неопределенное;
- когда у поведения нет никакого явного определения.

Первые два случая часто называют *явным неопределенным поведением*, а третий — *неявным неопределенным поведением*. Но по своему значению они эквивалентны; каждый из них описывает поведение, которое не определено. Приложение J.2 к стандарту C, «Неопределенное поведение», содержит перечень примеров такого поведения в C.

Разработчики часто неверно считают неопределенное поведение ошибками или упущениями в стандарте C, однако решение о том, считать ли поведение неопределенным, является *намеренным* и *взвешенным*. Комитет во главе стандарта C считает неопределенным поведение, которое:

- дает разработчику реализации C повод не отлавливать программные ошибки, которые сложно диагностировать;
- избегает определения запутанных, граничных случаев, которые хорошо подходят только для какой-то одной стратегии реализации;
- устанавливает рамки возможного расширения языка, в которых разработчик реализации C может дополнить язык, предоставив определение официально неопределенного поведения.

Эти три причины довольно сильно отличаются друг от друга, но все они относятся к проблемам переносимости. По ходу чтения книги вы будете сталкиваться с примерами всех трех вариантов. Компиляторы (реализации) могут делать следующее:

- полностью игнорировать неопределенное поведение, возвращая непредсказуемые результаты;
- вести себя задокументированным образом, характерным для окружения (с выводом диагностического сообщения или без него);
- прекратить компиляцию или выполнение (с выводом диагностического сообщения).

Все эти решения далеки от идеала (особенно первое), но неопределенного поведения лучше всего избегать, если только реализация не доопределяет это поведение с помощью расширения языка[1].

Поведение, зависящее от региональных параметров, и распространенные расширения

На *поведение, зависящее от региональных параметров (locale-specific behavior)*, влияют местные особенности, задокументированные в каждой отдельной реализации, включая национальные, культурные и языковые. Во многих системах широко применяются распространенные расширения, но их нельзя перенести на все реализации.

Резюме

В данной главе вы узнали, как написать, скомпилировать и запустить простую программу на языке C. Вслед за этим мы рассмотрели несколько редакторов, интерактивных сред разработки и компиляторов, с помощью которых можно разрабатывать программы на C для таких систем, как Windows, Linux и macOS. В целом рекомендуется использовать самые последние версии компиляторов и других инструментов, поскольку они обычно поддерживают более новые возможности языка программирования C, предоставляя лучшие диагностические сообщения и средства оптимизации. Если они нарушают работу существующего ПО или вы уже готовитесь развернуть свой код, то их лучше не использовать, чтобы не пришлось вносить изменения в хорошо протестированное приложение. Мы завершили эту главу обсуждением переносимости программ, написанных на C.

В следующих главах речь пойдет о конкретных возможностях языка C и его библиотеки. Начнем с объектов, функций и типов.

[1] У некоторых компиляторов есть «педантичный» режим, который может уведомлять программиста о таких проблемах переносимости.

2 Объекты, функции и типы

В этой главе вы познакомитесь с объектами, функциями и типами. Мы поговорим о том, как объявлять переменные (объекты с идентификаторами) и функции, как получать адреса объектов и разыменовывать их указатели. Вы уже видели некоторые типы, доступные программистам на C. Первым делом в данной главе вы узнаете то, что я усвоил чуть ли не в последнюю очередь: каждый тип в C является либо *объектным*, либо *функциональным*.

Объекты, функции, типы и указатели

Объект — это хранилище, в котором можно представлять значения. Если быть точным, то в стандарте C (ISO/IEC 9899:2018) объектом называется «область хранилища данных в среде выполнения, содержимое которого может представлять значения» с примечанием: «при обращении к объекту можно считать, что он обладает определенным типом». Один из примеров объекта — переменная.

Переменные имеют объявленный *тип*, который говорит о том, какого рода объект представляет его значение. Например, объект типа `int` содержит целочисленное значение. Важность типа объясняется тем, что набор битов, представляющий объект одного типа, скорее всего, будет иметь другое значение, если его интерпретировать как объект другого типа. Например, в IEEE 754 (стандарт IEEE для арифметических операций с плавающей запятой) число 1 представлено как `0x3f800000` (IEEE 754–2008). Но если интерпретировать тот же набор битов как целое число, то вместо 1 получится значение 1 065 353 216.

Функции не являются объектами, но тоже имеют тип. Тип функции характеризуется как ее возвращаемым значением, так и числом и типами ее параметров.

Кроме того, в C есть *указатели*, которые можно считать *адресами* — областями памяти, в которых хранятся объекты или функции. Тип указателя, основанный на типе функции или объекта, называется *ссылочным типом*. Указатель, имеющий ссылочный тип `T`, называют *указателем на* `T`.

Объекты и функции — это разные вещи, и потому объектные указатели отличаются от функциональных и их нельзя использовать как взаимозаменяемые. В следующем разделе вы напишете простую программу, которая пытается поменять местами значения двух переменных. Это поможет вам лучше разобраться в объектах, функциях, указателях и типах.

Объявление переменных

При объявлении переменной вы назначаете ей тип и даете ей имя — *идентификатор*, по которому к ней можно обращаться.

В листинге 2.1 объявляются два целочисленных объекта с исходными значениями. Эта простая программа также объявляет, но не определяет функцию `swap`, которая впоследствии будет менять эти значения местами.

Листинг 2.1. Программа, которая должна менять местами два целых числа

```
#include <stdio.h>

❶ void swap(int, int); // определена в листинге 2.2

int main(void) {
  int a = 21;
  int b = 17;

❷ swap(a, b);
  printf("main: a = %d, b = %d\n", a, b);
  return 0;
}
```

Эта демонстрационная программа состоит из функции `main` с единственным блоком кода между фигурными скобками. Такого рода блоки называют *составными операторами*. Внутри функции `main` мы определяем две переменные, `a` и `b`. Мы объявляем их как переменные типа `int`

и присваиваем им значения 21 и 17 соответственно. У всякой переменной должно быть объявление. Затем внутри `main` происходит вызов функции `swap` ❷, чтобы поменять местами значения этих двух целочисленных переменных. В данной программе функция `swap` объявлена ❶, но не определена. Позже в этом разделе мы рассмотрим некоторые потенциальные ее реализации.

Объявление нескольких переменных

Вы можете объявлять сразу несколько переменных, но это может сделать код запутанным, если они имеют разные типы или являются указателями или массивами. Например, все следующие объявления являются корректными:

```
char *src, c;
int x, y[5];
int m[12], n[15][3], o[21];
```

В первой строчке объявляются две переменные, `src` и `c`, которые имеют разные типы. Переменная `src` имеет тип `char *`, а `c` — тип `char`. Во второй строчке тоже происходит объявление двух переменных разных типов, `x` и `y`; первая имеет тип `int`, а вторая является массивом из пяти элементов типа `int`. В третьей строчке объявлено три массива, `m`, `n` и `o`, с разной размерностью и количеством элементов.

Эти объявления будет легче понять, если разделить их по отдельным строчкам:

```
char *src;    // src имеет тип char *
char c;       // c имеет тип char
int x;        // x имеет тип int
int y[5];     // y — это массив из 5 элементов типа int
int m[12];    // m — это массив из 12 элементов типа int
int n[15][3]; // n — это массив из 15 массивов, состоящих из трех
              // элементов типа int
int o[21];    // o — это массив из 21 элемента типа int
```

В удобочитаемом и понятном коде реже встречаются ошибки.

Перестановка значений местами (первая попытка)

У каждого объекта есть срок хранения, который определяет его *время жизни (lifetime)* — период выполнения программы, на протяжении которого этот объект существует, где-то хранится, имеет постоянный адрес и сохраняет последнее присвоенное ему значение. К объектам нельзя обращаться вне данного периода.

Локальные переменные, такие как `a` и `b` из листинга 2.1, имеют *автоматический срок хранения (storage duration)*; то есть они существуют, пока поток выполнения не покинет блок, в котором они определены. Попробуем поменять местами значения, хранящиеся в этих двух переменных.

Листинг 2.2 отображает нашу первую попытку реализовать функцию `swap`.

Листинг 2.2. Функция swap

```
void swap(int a, int b) {
  int t = a;
  a = b;
  b = t;
  printf("swap: a = %d, b = %d\n", a, b);
}
```

Функция `swap` объявляет два параметра, `a` и `b`, с помощью которых вы передаете ей аргументы. В C есть разница между *параметрами* и *аргументами*. Первые — это объекты, которые объявляются вместе с функцией и получают значения при входе в нее, а вторые — это выражения, разделяемые запятыми, которые указываются в выражении вызова функции. Мы также объявляем в функции `swap` временную переменную `t` типа `int` и инициализируем ее с помощью значения `a`. Данная переменная используется для временного хранения значения `a`, чтобы оно не было утеряно во время перестановки.

Теперь мы можем скомпилировать и проверить нашу полноценную программу, запустив сгенерированный исполняемый файл:

```
% ./a.out
swap: a = 17, b = 21
main: a = 21, b = 17
```

Результат может вас удивить. Изначально переменные `a` и `b` равны `21` и `17` соответственно. Первый вызов `printf` внутри функции `swap` показывает, что эти два значения поменялись местами, однако, если верить второму вызову `printf` в `main`, исходные значения остались неизменными. Посмотрим, что же произошло.

В языке C передача аргументов при вызове происходит *по значению*; то есть когда вы предоставляете функции аргумент, его значение копируется в отдельную переменную, доступную для использования внутри этой функции. Функция `swap` присваивает значения объектов, переданных

в виде аргументов, соответствующим параметрам. Изменение значений параметров в функции не влияет на значения в вызывающем коде, поскольку это разные объекты. Следовательно, переменные `a` и `b` сохраняют исходные значения в `main` во время второго вызова `printf`. Программа должна была поменять местами значения этих двух объектов. Протестировав ее, мы обнаружили в ней ошибку (или дефект).

Перестановка значений местами (вторая попытка)

Чтобы исправить эту ошибку, мы можем переписать функцию `swap` с помощью указателей. Применим операцию косвенного обращения (или разыменовывания) `*`, чтобы объявить и разыменовать указатели, как показано в листинге 2.3.

Листинг 2.3. Переработанная функция swap с использованием указателей

```
void swap(int *pa, int *pb) {
  int t = *pa;
  *pa = *pb;
  *pb = t;
  return;
}
```

В объявлении или определении функции операция `*` выступает частью объявления указателя, сигнализируя о том, что параметр является указателем на объект или функцию заданного типа. В переписанной функции `swap` указано два параметра, `pa` и `pb`, объявленных как указатели типа `int`.

При использовании унарной операции `*` в выражениях внутри функции она разыменовывает указатель на объект. Например, взгляните на следующую операцию присваивания:

```
pa = pb;
```

Здесь значение указателя `pa` заменяется значением указателя `pb`. Теперь посмотрите, как происходит присваивание в функции `swap`:

```
*pa = *pb;
```

Это выражение разыменовывает указатель `pb`, считывает значение, на которое тот ссылается, разыменовывает указатель `pa` и затем вместо

значения по адресу, на который ссылается `pa`, записывает значение, на которое ссылается `pb`.

При вызове функции `swap` в `main` необходимо также указать амперсанд (`&`) перед именем каждой переменной:

```
swap(&a, &b);
```

Унарная операция `&` используется для *взятия адреса*. Она генерирует указатель на свой операнд. Это изменение необходимо, поскольку функция `swap` теперь принимает в качестве параметров указатели на объекты типа `int`, а не просто значения данного типа.

В листинге 2.4 показана вся программа `swap` целиком, с описанием объектов, создаваемых во время ее работы, и их значений.

Листинг 2.4. Имитация передачи аргументов по ссылке

```
#include <stdio.h>
void swap(int *pa, int *pb) {   // pa → a: 21 pb → b: 17
  int t = *pa;                  // t: 21
  *pa = *pb;                    // pa → a: 17 pb → b: 17
  *pb = t;                      // pa → a: 17 pb → b: 21

}
int main(void) {
  int a = 21;                   // a: 21
  int b = 17;                   // b: 17
  swap(&a, &b);
  printf("a = %d, b = %d\n", a, b);   // a: 17 b: 21
  return 0;
}
```

Во время входа в блок `main` переменным `a` и `b` присваиваются значения `21` и `17` соответственно. Затем код берет адреса этих объектов и передает их функции `swap` в качестве аргументов.

Параметры `pa` и `pb` внутри `swap` объявлены в виде указателей типа `int` и содержат копии аргументов, переданных этой функции из вызывающего кода (в данном случае `main`). Эти копии адресов по-прежнему ссылаются на те же объекты, поэтому, когда функция `swap` меняет эти объекты местами, содержимое исходных объектов, объявленных в `main`, тоже меняется. Данный подход имитирует передачу аргументов *по ссылке*: сначала генерируются адреса объектов, которые передаются по значению, а затем они разыменовываются для доступа к исходным объектам.

Область видимости

Объекты, функции, макросы и другие идентификаторы языка C имеют *область видимости*, которая определяет, где к ним можно обращаться. В языке C область видимости может охватывать файл, блок, прототип функции и саму функцию.

Область видимости объектного или функционального идентификатора определяется тем, где он объявлен. Если объявление сделано за пределами какого-либо блока или списка параметров, то идентификатор доступен *на уровне файла*; это значит, что область видимости охватывает весь файл, в котором он находится, а также любые другие файлы, подключенные после его объявления.

Если объявление происходит внутри блока или списка параметров, то оно имеет *блочную область видимости*; то есть объявленный им идентификатор доступен только внутри этого блока. Примерами этого выступают идентификаторы `a` и `b` из листинга 2.4; они позволяют ссылаться на эти переменные только внутри блока кода в функции `main`, в котором они объявлены.

Если объявление происходит внутри списка параметров прототипа функции (но не в ее теле), то область видимости будет ограничена ее объявлением (то есть прототипом). *Область видимости функции* охватывает ее определение между открывающей (`{`) и закрывающей (`}`) скобками. Единственным видом идентификаторов, который имеет такую область видимости, являются *метки* — идентификаторы, завершающиеся двоеточием и определяющие оператор в функции, к которому может перейти управление. О метках и передаче управления речь пойдет в главе 5.

Области видимости могут быть *вложенными* и находиться как *внутри*, так и *снаружи* относительно других областей видимости. Например, у вас может быть одна блочная область видимости внутри другой, и все они определены на уровне файла. Внутренняя область имеет доступ к наружной, но не наоборот. Как понятно из названия, любая внутренняя область видимости не может выходить за пределы наружных, которые ее охватывают.

Если объявить один и тот же идентификатор как во внутренней, так и в наружной областях видимости, то внутренняя версия *перекрывает* наружную, имея более высокий приоритет. В данном случае имя идентификатора

будет ссылаться на объект во внутренней области видимости; объект из наружной области скрывается, и на него нельзя сослаться по имени. Чтобы это не создавало проблем, лучше использовать разные имена.

В листинге 2.5 показаны разные области видимости и то, как идентификаторы, объявленные во внутренних областях, могут перекрывать идентификаторы, объявленные в наружных.

Листинг 2.5. Области видимости

```
int j; // начинается область видимости уровня файла j

void f(int i) {         // начинается блочная область видимости i
  int j = 1;            // начинается блочная область видимости j
                        // перекрывает j в области видимости уровня файла
  i++;                  // i ссылается на параметр функции
  for (int i = 0; i < 2; i++) { // начинается блочная область видимости i
                                // внутри цикла
    int j = 2;          // начинается блочная область видимости j
                        // перекрывает внешнюю j
    printf("%d\n", j); // внутренняя j в области видимости, выводит 2
  }                     // заканчивается блочная область видимости
                        // внутренних i и j
  printf("%d\n", j);    // внешняя j в области видимости, выводит 1
}                       // заканчивается блочная область видимости i и j

void g(int j);          // j имеет область видимости уровня прототипа
                        // перекрывает j уровня файла
```

С этим кодом все в порядке, если предположить, что комментарии правильно передают замысел автора. Чтобы избежать путаницы и потенциальных ошибок, для разных идентификаторов лучше использовать разные имена. Применение коротких имен наподобие `i` и `j` подходит для идентификаторов с маленькими областями видимости. Если область видимости большая, то лучше прибегнуть к наглядным именам, которые вряд ли будут перекрыты во вложенных областях. Некоторые компиляторы предупреждают о перекрытых идентификаторах.

Срок хранения

Объекты имеют срок хранения, который определяет их время жизни. В целом сроки хранения бывают четырех видов: автоматические, статические, потоковые и выделенные. Вы уже видели, что объекты с автоматическим сроком хранения объявляются внутри блока или в качестве параметров

функции. Их время жизни начинается в момент выполнения блока, в котором они объявлены, и заканчивается при его завершении. Если блок вызывается рекурсивно, то при каждом вызове создается новый объект с собственным сроком хранения.

ПРИМЕЧАНИЕ

> Область видимости и время жизни — совершенно разные понятия. Первое относится к идентификаторам, а второе — к объектам. Областью видимости идентификатора является участок кода, в котором к объекту, обозначенному этим идентификатором, можно получить доступ по его имени. Время жизни объекта — это период, на протяжении которого он существует.

Объекты, объявленные на уровне файла, имеют *статический* срок хранения. Их время жизни охватывает весь период выполнения программы, а значения, которые в них хранятся, инициализируются еще до ее запуска. Вы также можете объявить переменную со статическим сроком хранения внутри блочной области видимости, используя спецификатор класса хранения `static`, как показано в следующем примере со счетчиком (листинг 2.6). Эти объекты продолжают существовать после завершения функции.

Листинг 2.6. Пример счетчика

```
void increment(void) {
  static unsigned int counter = 0;
  counter++;
  printf("%d ", counter);
}

int main(void) {
  for (int i = 0; i < 5; i++) {
    increment();
  }
  return 0;
}
```

Эта программа выводит `1 2 3 4 5`. Мы присваиваем статической переменной `counter` значение `0` во время запуска программы и затем инкрементируем его при каждом вызове функции `increment`. Время жизни `counter` охватывает весь период выполнения программы, а последнее значение данной переменной будет храниться, пока она существует. То же по-

ведение можно получить, объявив `counter` на уровне файла. Однако при разработке ПО рекомендуется как можно сильнее ограничивать область видимости объектов.

Статические объекты должны инициализироваться с помощью константного значения, а не переменной:

```
int *func(int i) {
  const int j = i;  // правильно
  static int k = j; // ошибка
  return &k;
}
```

К константным значениям относятся константы-литералы (например, `1`, `'a'` или `0xFF`), члены `enum` и результаты работы операций, таких как `alignof` или `sizeof`, но не объекты со спецификатором `const`.

Потоковый срок хранения используется в конкурентном программировании и не рассматривается в этой книге. *Динамический* срок хранения относится к динамически выделяемой памяти и обсуждается в главе 6.

Выравнивание

Типы объектов имеют требования к выравниванию, ограничивающие адреса, которые могут выделяться для объектов этих типов. *Выравнивание (alignment)* определяет количество байтов между смежными адресами, в которых может быть сохранен заданный объект. Процессор может поразному обращаться к выровненным и невыровненным данным (адрес выровненных данных может быть, например, кратен их размеру).

Некоторые машинные инструкции могут выполнять многобайтный доступ к данным, не выровненным по границе машинного слова, но это может отрицательно сказаться на производительности. Машинное слово — естественная единица фиксированного размера для измерения данных, с которой работают инструкции или аппаратные механизмы процессора. Некоторые платформы не поддерживают доступ к невыровненной памяти. Требования к выравниванию могут зависеть от размера машинного слова в конкретном процессоре (обычно это 16, 32 или 64 бита).

В целом программистам на C не нужно беспокоиться об этих требованиях, поскольку компилятор самостоятельно выбирает для своих

различных типов подходящее выравнивание. Гарантируется, что память, выделенная с помощью `malloc`, будет достаточно выровненной для всех стандартных типов, включая массивы и структуры. Но в редких случаях решения компилятора, используемые по умолчанию, приходится переопределять — например, чтобы выровнять данные на границах строк кэша, адреса должны быть кратны степеням двойки. Для выполнения этих требований традиционно задействуются команды компоновки, выделение дополнительной памяти с помощью `malloc` с последующим округлением пользовательского адреса вверх или другие операции с применением нестандартных средств.

В C11 появился простой способ указания выравнивания, обладающий совместимостью с будущими версиями стандарта. Выравнивание представляется как неотрицательное целое число типа `size_t`. Корректное выравнивание должно быть степенью двойки. В листинге 2.7 спецификатор выравнивания позволяет убедиться в том, что значение `good_buff` как следует выровнено (у `bad_buff` может быть неправильное выравнивание для выражений доступа к членам).

Листинг 2.7. Использование ключевого слова _Alignas[1]

```
struct S {
  int i; double d; char c;
};

int main(void) {
  unsigned char bad_buff[sizeof(struct S)];
  _Alignas(struct S) unsigned char good_buff[sizeof(struct S)];

  struct S *bad_s_ptr = (struct S *)bad_buff;
  // неправильное выравнивание указателя
  struct S *good_s_ptr = (struct S *)good_buff;
  // правильное выравнивание указателя
}
```

Способы выравнивания упорядочены от слабых к сильным (строгим). Чем строже выравнивание, тем больше его значение. Адрес, удовлетворяющий требованиям строгого выравнивания, соответствует требованиям любого корректного выравнивания, которое является более слабым.

[1] В этом примере массивы не инициализированы, поэтому разыменовывать указатели bad_s_ptr и good_s_ptr нельзя в любом случае, это неопределенное поведение. — *Примеч. науч. ред.*

Объектные типы

В этом разделе рассматриваются объектные типы, доступные в C. В частности, мы обсудим логические, символьные и численные типы (включая целые и числа с плавающей запятой).

Логические типы

Объекты, объявленные как `_Bool`, могут хранить либо 0, либо 1. *Логический* (или *булев*) тип впервые появился в C99. Он начинается с подчеркивания, чтобы его не путали с пользовательскими идентификаторами `bool` или `boolean`, объявленными в существующих программах. Если название идентификатора начинается с подчеркивания, за которым идет либо большая буква, либо еще одно подчеркивание, то оно зарезервировано. Это сделано для того, чтобы комитет по стандартизации C мог создавать новые ключевые слова наподобие `_Bool`. Предполагается, что вы избегаете применения зарезервированных идентификаторов; в противном случае сами виноваты — внимательно читайте стандарт.

Подключив заголовочный файл `<stdbool.h>`, вы сможете записывать этот идентификатор как `bool` и присваивать ему значения `true` (разворачивается в целочисленную константу `1`) и `false` (разворачивается в целочисленную константу `0`). Ниже мы объявляем две логические переменные, используя оба названия типа:

```
#include <stdbool.h>
_Bool flag1 = 0;
bool flag2 = false;
```

Оба варианта будут работать, но лучше использовать `bool`, так как именно ему отдается предпочтение в долгосрочной перспективе развития языка C.

Символьные типы

Язык C определяет три *символьных типа*: `char`, `signed char` и `unsigned char`. В любой реализации компилятора `char` имеет те же выравнивание, размер, диапазон, представление и поведение, что и `signed char` или `unsigned char`. Тем не менее `char` — отдельный тип, несовместимый с двумя другими.

Для представления символьных данных в программах на языке C обычно используется тип `char`. В частности, объекты этого типа должны быть

способны хранить базовый минимальный набор символов, необходимый среде выполнения, включая прописные и строчные латинские буквы, десять десятичных цифр, пробельные символы, различные знаки препинания и управляющие символы. Тип `char` не предназначен для целочисленных данных; для небольших целых значений со знаком или без более безопасно использовать `signed char` или `unsigned char` соответственно.

Базовый минимальный набор символов подходит для многих традиционных задач по обработке данных, но отсутствие в нем букв, не входящих в английский алфавит, затрудняет его использование в других странах. Чтобы решить эту проблему, комитет во главе стандарта C определил новый, широкий тип с поддержкой больших наборов символов. Представлять *широкие символы* из более объемных наборов символов можно с помощью типа `wchar_t`, который обычно занимает больше места, чем базовые символы. В разных реализациях языка широкие символы, как правило, занимают 16 или 32 бита. Стандартная библиотека C предоставляет функции с поддержкой как узких, так и широких символьных типов.

Численные типы

Язык C поддерживает несколько *численных типов*, которые можно использовать для представления целых чисел, перечислителей и значений с плавающей запятой. Некоторые из них более подробно рассматриваются в главе 3, а здесь дается лишь краткое введение.

Целочисленные типы

Целочисленные типы со знаком можно использовать для представления отрицательных, положительных и нулевых значений. В их число входят `signed char`, `short int`, `int`, `long int` и `long long int`.

При объявлении значений этих типов ключевое слово `int` можно опускать (если только это не сам тип `int`), что позволяет вместо `long long int`, к примеру, написать `long long`.

Для каждого целочисленного типа со знаком предусмотрен соответствующий *беззнаковый целочисленный тип*, который занимает столько же места: `unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int` и `unsigned long long int`. С помощью беззнаковых типов можно представлять только положительные и нулевые значения.

Целочисленные типы, со знаком и без, используются для представления целых чисел разной длины. Размер каждого из них определяется конкретной платформой с учетом некоторых ограничений. Каждый тип имеет минимальный гарантированный диапазон. Типы упорядочены по ширине; это гарантирует, что *более широкий* тип как минимум не уступает по размерам *более узкому*. Благодаря этому объект типа `long long int` может представить любые значения объекта типа `long int`, а объект типа `long int` вмещает в себя все значения, которые могут быть представлены объектом типа `int`, и т. д. Конкретный размер разных целочисленных типов можно определить по минимально и максимально допустимым значениям, указанным в заголовочном файле `<limits.h>`.

Тип `int` обычно имеет естественный размер, вытекающий из архитектуры среды выполнения. Таким образом, в 16-битной архитектуре это 16 бит, а в 32-битной — 32 бита. Выделить целочисленные значения нужного вам размера можно с помощью определения типов наподобие `uint32_t` из заголовочного файла `<stdint.h>` или `<inttypes.h>`. В этих файлах содержатся определения самых широких целочисленных типов, которые доступны на конкретной платформе: `uintmax_t` и `intmax_t`.

В главе 3 целочисленные типы рассматриваются в мельчайших подробностях.

Перечисляемые типы

Перечисление (`enum`, от enumeration) позволяет определить тип, который назначает имена *(перечислители)* целочисленным значениям в ситуациях, когда требуется перечисляемый набор постоянных значений. Ниже показаны примеры перечислений:

```
enum day { sun, mon, tue, wed, thu, fri, sat };
enum cardinal_points { north = 0, east = 90, south = 180, west = 270 };
enum months { jan = 1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov,
dec };
```

Если не присвоить значение первому перечислителю с помощью операции `=`, то его перечисляемая константа будет равна 0, и каждый следующий перечислитель без знака `=` будет прибавлять 1 к предыдущей константе. Следовательно, значение `sun` в перечислении `day` равно 0, `mon` равно 1 и т. д.

Значения для перечислителей можно выбирать самостоятельно, как в случае с `cardinal_points`. Использование операции `=` может привести

к повторению значений, что чревато проблемами, если вы ошибочно предполагаете, будто все значения должны быть уникальными. В перечислении `month` первому перечислителю назначается 1, а значение каждого следующего перечислителя, который не задан явно, инкрементируется на 1.

Значения констант в перечислении должны быть совместимы с типом `int`, хотя в действительности их тип зависит от реализации. Например, Visual C++ использует `signed int`, а GCC — `unsigned int`.

Типы с плавающей запятой

Язык C поддерживает три *типа с плавающей запятой*: `float`, `double` и `long double`. Арифметические операции с плавающей запятой похожи на операции с числами с плавающей запятой и зачастую служат для них моделью. Стандарт языка C допускает разные варианты представлений чисел с плавающей запятой, и в большинстве систем реализован соответствующий стандарт IEEE (IEEE 754–2008). Выбор того или иного варианта зависит от реализации. Типы с плавающей запятой будут подробно рассмотрены в главе 3.

Тип void

Тип `void` довольно необычен. Ключевое слово `void` (само по себе) означает «не может содержать никаких значений». Например, с его помощью можно сигнализировать о том, что ваша функция не возвращает значение или не принимает аргументов (если указать его в качестве единственного параметра). С другой стороны, *производный тип* `void *` означает, что указатель может ссылаться на *любой* объект. Мы обсудим производные типы позже в этой главе.

Функциональные типы

Функциональные типы являются производными. В данном случае они основаны на типе возвращаемого значения, а также на количестве и типах параметров функции. Тип возвращаемого значения не может быть массивом.

При объявлении функции нужно указать ее имя и тип значения, которое она возвращает. Если объявление совмещено с определением и предусматривает список параметров, в объявлении каждого параметра необходимо

указать идентификатор. Исключение составляет случай, когда имеется только один параметр типа `void`, который не нуждается в идентификаторе.

Ниже показано несколько объявлений функциональных типов:

```
int f(void);
int *fip();
void g(int i, int j);
void h(int, int);
```

Сначала мы объявляем функцию `f` без параметров, которая возвращает `int`. Далее объявляется функция `fip` без списка параметров; она возвращает указатель на `int`. В конце мы видим объявления двух функций, `g` и `h`, каждая из которых возвращает `void` и принимает по два параметра типа `int`.

Задать параметры с идентификаторами (как в случае с `g`) может быть проблематично, если идентификатором выступает макрос. Однако предоставление именованных параметров способствует написанию самодокументируемого кода, поэтому опускать идентификаторы (как это сделано в `h`) обычно не рекомендуется.

В объявлении функции перечислять параметры не обязательно. Но если этого не делать, то можно столкнуться с проблемами. Если бы вы записали объявление для `fip` в C++, то у вас получилась бы функция, которая не принимает аргументов и возвращает `int *`. В языке C эта же функция имеет тот же возвращаемый тип, но принимает любое количество параметров любого типа. В C никогда не следует объявлять функции с пустым списком параметров. Во-первых, это устаревшая возможность языка, которая в будущем может исчезнуть. Во-вторых, ваш код может быть перенесен в C++, поэтому явно перечисляйте типы параметров и используйте `void`, если функция ничего не принимает.

Сочетание типа функции и списка с типами ее параметров является ее *прототипом*. Он сообщает компилятору о количестве и типах параметров, которые она принимает. С помощью этой информации компиляторы следят за тем, чтобы в *определении функции* и в любых ее вызовах использовалось корректное количество параметров с подходящими типами.

Определение функции содержит ее непосредственную реализацию. Взгляните на следующее определение:

```
int max(int a, int b)
{ return a > b ? a : b; }
```

Элемент `int` — это спецификатор возвращаемого значения, `max(int a, int b)` — ее объявление, а `{ return a > b ? a : b; }` — ее тело. Задавая тип функции, нельзя указывать никакие квалификаторы (см. раздел «Квалификаторы типов» на с. 61). В самом теле функции используется условная операция (`? :`), которая более подробно будет рассмотрена в главе 4. Данное выражение означает следующее: *если* `a` *больше* `b`, *то вернуть* `a`; *в противном случае вернуть* `b`.

Производные типы

Производными называют типы, основанные на других типах. В их число входят указатели, массивы, определения типов, структуры и объединения. О них обо всех мы поговорим в этом разделе.

Указатели

Тип указателя является производным от типа функции или объекта, на которые он указывает. Его также называют *ссылочным* типом. Указатель предоставляет ссылку на сущность ссылочного типа.

Ниже объявлены три указателя на `int`, `char` и `void`:

```
int *ip;
char *cp;
void *vp;
```

Ранее в этой главе вы познакомились с операциями получения адреса (`&`) и косвенного обращения (`*`). Операция `&` позволяет получить адрес объекта или функции. Например, если объект имеет тип `int`, то эта операция вернет указатель типа `int`:

```
int i = 17;
int *ip = &i;
```

Мы объявляем переменную `ip` в качестве указателя на `int` и присваиваем ей адрес `i`. Вы также можете применить операцию `&` к результату выполнения операции *:

```
ip = &*ip;
```

В результате разыменования `ip` с помощью операции косвенного обращения мы получаем сам объект `i`. Адрес `*ip`, полученный с помощью `&`,

является указателем, поэтому эти две операции компенсируют друг друга.

Унарная операция * возвращает значение, на которое ссылается указатель. Она обозначает *косвенность* и работает только с указателями. Если операнд указывает на функцию, то результатом использования * будет ее обозначение (designator), а если на объект, то результатом будет значение этого объекта. Например, если операнд является указателем на `int`, то операция косвенного обращения вернет тип `int`. Если указатель не ссылается на действительный объект или функцию, то может произойти что-то плохое.

Массивы

Массив — это последовательность объектов одного и того же типа, выделенных один за другим. Тип массива характеризуется типом и количеством его элементов. Ниже мы объявляем массив из 11 элементов типа `int` с идентификатором `ia` и массив из 17 указателей на `float` с идентификатором `afp`:

```
int ia[11];
float *afp[17];
```

Квадратные скобки (`[]`) используются для идентификации элемента массива. Например, следующий демонстрационный фрагмент кода создает строку `"0123456789"`, чтобы показать, как присваивать значения элементам массива:

```
char str[11];
for (unsigned int i = 0; i < 10; ++i) {
❶ str[i] = '0' + i;
}
str[10] = '\0';
```

В первой строчке объявляется массив типа `char` размером 11. В результате выделяется место, достаточное для хранения десяти символов плюс `\0` (также известный как «нуль-символ»). Цикл `for` повторяется десять раз, а значение `i` изменяется от 0 до 9. На каждой итерации результат выражения `'0' + i` присваивается элементу `str[i]`. После завершения цикла нуль-символ копируется в заключительный элемент массива `str[10]`.

В выражении ❶ `str` автоматически преобразуется в указатель на первый элемент массива (объект типа `char`), а `i` имеет беззнаковый целочисленный

тип. Благодаря наличию операций индексации (`[]`) и сложения (`+`) выражение `str[i]` идентично `*(str + i)`. Когда `str` является массивом (как в данном случае), `str[i]` обозначает его `i`-й элемент (начиная с 0). Поскольку массивы индексируются с 0, элементы `char str[11]` пронумерованы от 0 до 10; как видно в последней строчке этого примера, `10` — последний элемент.

Если операнд унарной операции `&` является результатом операции `[]`, то его можно было бы получить, убрав `&` и заменив `[]` операцией `+`. Например, выражение `&str[10]` эквивалентно `str + 10`.

Вы также можете объявлять многомерные массивы. В листинге 2.8 показана функция `main`, в которой объявляется двумерный массив `arr` типа `int` размером 5×3, который еще называют *матрицей*.

Листинг 2.8. Операции с матрицей

```
void func(int arr[5]);
int main(void) {
  unsigned int i = 0;
  unsigned int j = 0;
  int arr[3][5];
❶ func(arr[i]);
❷ int x = arr[i][j];
  return 0;
}
```

Если быть более точным, то `arr` — это массив из трех элементов, каждый из которых является массивом из пяти элементов типа `int`. Когда вы используете выражение `arr[i]` ❶ (эквивалентное `*(arr+i)`), происходит следующее.

1. Массив `arr` преобразуется в указатель на исходный массив из пяти элементов типа `int`, начиная с `arr[i]`.
2. Индекс `i` подстраивается под тип `arr` за счет умножения на размер одного массива из пяти объектов `int`.
3. Результаты первых двух шагов складываются.
4. Происходит косвенное обращение к результату, чтобы получить массив из пяти элементов типа `int`.

При использовании выражения `arr[i][j]` ❷ массив преобразуется в указатель на первый элемент типа `int`, поэтому `arr[i][j]` возвращает объект типа `int`.

Определения типов

Ключевое слово `typedef` используется для создания псевдонима существующего типа; оно никогда не создает новый тип. Например, в каждом из следующих объявлений создается новый псевдоним типа:

```
typedef unsigned int uint_type;
typedef signed char schar_type, *schar_p, (*fp)(void);
```

В первой строчке мы объявляем `uint_type` в качестве псевдонима для типа `unsigned int`. Во второй строчке объявляем псевдонимы `schar_type`, `schar_p` и `fp` для типов `signed char`, `signed char *` и `signed char(*)(void)` соответственно. Идентификаторы, находящиеся в стандартных заголовочных файлах и заканчивающиеся на `_t`, являются определениями типов (псевдонимами для существующих типов). В целом вы не должны соблюдать это соглашение об именовании в собственном коде, поскольку стандарт C резервирует идентификаторы, соответствующие шаблонам `int[0-9a-z_]*_t` и `uint[0-9a-z_]*_t`, а спецификация POSIX (Portable Operating System Interface — переносимый интерфейс операционных систем) резервирует все идентификаторы, которые заканчиваются на `_t`. В случае определения идентификатора с таким именем может возникнуть конфликт с именами, которые используются реализацией. Это, в свою очередь, может спровоцировать проблемы, которые будет сложно диагностировать.

Структуры

Структура (или `struct`) содержит последовательно выделенные объекты-члены. Каждый объект обладает собственным именем и может иметь отдельный тип — для сравнения, в массивах все элементы должны быть одного типа. Структуры похожи на тип данных *«запись»*, который встречается в других языках программирования. В листинге 2.9 объявляется объект с идентификатором `sigline` и типом `struct sigrecord`, а указатель на объект типа `struct sigrecord` имеет идентификатор `sigline_p`.

Листинг 2.9. Структура sigrecord

```
struct sigrecord {
  int signum;
  char signame[20];
  char sigdesc[100];
} sigline, *sigline_p;
```

У этой структуры есть три объекта-члена: `signum` (объект типа `int`), `signame` (массив из 20 элементов типа `char`) и `sigdesc` (массив из 100 элементов типа `char`).

Структуры позволяют объявлять коллекции связанных между собой объектов и могут использоваться для представления таких понятий, как дата, клиент или личное дело работника. Они особенно полезны для группирования объектов, которые часто передаются вместе в виде аргументов функций; это избавит вас от необходимости постоянно передавать каждый объект по отдельности.

Определив структуру, вы, вероятно, захотите обращаться к ее членам. Для обращения к членам объекта структурных типов служит операция доступа (`.`). Если у вас есть указатель на структуру, то для обращения к ее членам предусмотрена операция `->`. Использование каждой из этих операций показано в листинге 2.10.

Листинг 2.10. Обращение к членам структуры

```
sigline.signum = 5;
strcpy(sigline.signame, "SIGINT");
strcpy(sigline.sigdesc, "Interrupt from keyboard");

❶ sigline_p = &sigline;

sigline_p->signum = 5;
strcpy(sigline_p->signame, "SIGINT");
strcpy(sigline_p->sigdesc, "Interrupt from keyboard");
```

В первых трех строчках листинга 2.10 происходит прямой доступ к членам объекта `sigline` с помощью операции доступа (`.`). В строчке ❶ мы присваиваем указателю `sigline_p` адрес объекта `sigline`. В заключительных трех строчках программы мы косвенно обращаемся к членам объекта `sigline` через указатель `sigline_p`, используя операцию `->`.

Объединения

Объединения похожи на структуры, только их члены используют одну и ту же память. В один момент времени объединение может содержать объект одного типа, а в другой момент времени — объект другого типа, но никогда не может содержать оба объекта сразу. Объединения используются в основном для экономии памяти.

В листинге 2.11 показано объединение `u`, которое содержит три структуры: `n`, `ni` и `nf`. Его можно задействовать в дереве, графе и других структурах данных с узлами, которые могут содержать как целочисленные (`ni`), так и вещественные значения (`nf`).

Листинг 2.11. Объединения

```
union {
  struct {
    int type;
  } n;
  struct {
    int type;
    int intnode;
  } ni;
  struct {
    int type;
    double doublenode;
  } nf;
} u;
u.nf.type = 1;
u.nf.doublenode = 3.14;
```

Как и в случае со структурами, доступ к членам объединения осуществляется с помощью операции доступа (`.`). Если у вас есть указатель на объединение, то для обращения к его членам предусмотрена операция `->`. В представленном выше листинге 2.11 член структуры `nf` с именем `type` доступен в виде `u.nf.type`, а доступ к члену `doublenode` происходит с помощью выражения `u.nf.doublenode`. Код, использующий такое объединение, обычно выясняет тип узла, проверяя значение, хранящееся в `u.n.type`, и затем, в зависимости от результата, обращается либо к `intnode`, либо к `doublenode`. Если бы это было реализовано в виде структуры, то в каждом узле было бы выделено место как для `intnode`, так и для `doublenode`. Использование объединения позволяет задействовать одно и то же место для обоих членов.

Теги

Теги — это специальный механизм именования структур, объединений и перечислений. Например, идентификатор `s`, представленный в следующей структуре, является тегом:

```
struct s {
  //---snip---
};
```

Сам по себе тег не является именем типа, и его нельзя использовать для объявления переменных (Сакс, 2002). Вместо этого переменные таких типов следует объявлять следующим образом:

```
struct s v;  // экземпляр структуры s
struct s *p; // указатель на структуру s
```

Названия объединений и перечислений — тоже теги, а не типы. Это значит, их недостаточно для объявления переменной. Например:

```
enum day { sun, mon, tue, wed, thu, fri, sat };
day today;         // ошибка
enum day tomorrow; // правильно
```

Теги структур, объединений и перечислений определяются в собственном *пространстве имен*, отдельно от обычных идентификаторов. Благодаря этому программа на C может иметь в одной области видимости тег и другой идентификатор, который выглядит точно так же:

```
enum status { ok, fail }; // перечисление
enum status status(void); // функция
```

Вы даже можете объявить объект `s` типа `struct s`:

```
struct s s;
```

Это не самое лучшее решение, но в языке C оно является корректным. Теги структур можно считать именами типов и определять для них псевдонимы с помощью `typedef`. Например:

```
typedef struct s { int x; } t;
```

Теперь вы можете объявлять переменные типа `t`, а не `struct s`. Имя тега в `struct`, `union` и `enum` указывать не обязательно, поэтому можете его опустить:

```
typedef struct { int x; } t;
```

Это будет работать всегда, кроме тех случаев, когда структура содержит указатель на саму себя:

```
struct tnode {
  int count;
  struct tnode *left;
  struct tnode *right;
};
```

Если опустить тег в первой строчке, то компилятору это может не понравиться, поскольку структура, упоминаемая в третьей и четвертой строчках, еще не объявлена или просто потому, что нигде не используется. Таким образом, вам не остается ничего другого, как объявить для данной структуры тег. Но вы также можете объявить `typedef`:

```
typedef struct tnode {
  int count;
  struct tnode *left;
  struct tnode *right;
} tnode;
```

Большинство программистов на C используют для тегов и `typedef` разные имена, хотя можно обойтись и общим именем. Вы также можете определить данный тип перед структурой; это позволит вам объявить члены `left` и `right`, которые ссылаются на другие объекты типа `tnode`:

```
typedef struct tnode tnode;
struct tnode {
  int count;
  tnode *left
  tnode *right;
} tnode;
```

Определения типов могут сделать код более понятным не только в случае их использования со структурами. Например, во всех трех объявлениях функции `signal`, представленных ниже, указан один и тот же тип:

```
typedef void fv(int), (*pfv)(int);
void (*signal(int, void (*)(int)))(int);
fv *signal(int, fv *);
pfv signal(int, pfv);
```

Квалификаторы типов

Все типы, рассмотренные до сих пор, были неквалифицированными. Тип можно сделать *квалифицированным* за счет использования одного или нескольких квалификаторов, таких как `const`, `volatile` и `restrict`. Каждый квалификатор влияет на поведение кода при обращении к объектам квалифицированного типа.

Квалифицированная и неквалифицированная версии типа являются взаимозаменяемыми в случае использования в качестве аргументов функций, возвращаемых значений и членов объединений.

ПРИМЕЧАНИЕ

Квалификатор типа _Atomic, появившийся в C11, предоставляет поддержку конкурентного выполнения.

Квалификатор const

Объекты, объявленные с использованием квалификатора `const` (`const`-квалифицированные объекты), не подлежат изменению. После инициализации им больше нельзя присваивать значения. Это значит, что компилятор может разместить их в памяти, доступной только для чтения, и любая попытка записи в них приведет к ошибке времени выполнения:

```
const int i = 1; // константа int
i = 2; // ошибка: i — это константа
```

Иногда можно случайно заставить компилятор внести изменение в тип с квалификатором `const`. В следующем примере мы берем адрес `const`-квалифицированного объекта `i` и сообщаем компилятору, что в действительности это указатель на `int`:

```
const int i = 1; // объект неконстантного типа
int *ip = (int *)&i;
*ip = 2;  // неопределенное поведение
```

Во время приведения типов язык C не позволяет избавиться от квалификатора `const`, если он использовался в исходном объекте. Этот код может показаться рабочим, но имеет дефект, который позже может привести к сбою. Например, компилятор может поместить `const`-квалифицированный объект в память, доступную только для чтения, и во время выполнения при попытке сохранить в него значение произойдет ошибка.

Язык C позволяет изменять объект, на который ссылается указатель с квалификатором `const`, за счет избавления от последнего в ходе приведения типов. При этом данный квалификатор не должен использоваться при объявлении исходного объекта:

```
int i = 12;
const int j = 12;
const int *ip = &i;
const int *jp = &j;
*(int *)ip = 42; // правильно
*(int *)jp = 42; // неопределенное поведение
```

Квалификатор volatile

Объекты `volatile`-квалифицированных типов имеют специальное назначение. Статические объекты с квалификатором `volatile` используются для представления портов ввода/вывода, отображенных в память, а статические объекты сразу с двумя квалификаторами, `const` и `volatile`, могут представлять порты ввода, отображенные в память, такие как часы реального времени.

Значения, хранящиеся в этих объектах, могут изменяться без ведома компилятора. Например, значение часов реального времени может меняться при каждом обращении, даже если сама программа это значение не меняет. Использование `volatile`-квалифицированного типа сообщает компилятору о том, что значение может измениться, и гарантирует, что код будет каждый раз обращаться к часам реального времени (в противном случае эти обращения могут быть удалены в ходе оптимизации или заменены уже считанным и закэшированным значением). Например, при обработке следующего кода компилятор должен сгенерировать инструкции, которые будут читать значение из `port` и затем записывать его обратно в `port`:

```
volatile int port;
port = port;
```

Если бы не было квалификатора `volatile`, то компилятор воспринял бы этот оператор как фиктивный (такой, который ничего не делает) и мог бы удалить как чтение, так и запись.

`volatile`-квалифицированные типы также используются для взаимодействия с обработчиками сигналов и операторами `setjmp`/`longjmp` (о тех и других можно прочесть в стандарте C). В отличие от Java и других языков программирования в C *не следует* применять `volatile`-квалифицированные типы для синхронизации между потоками выполнения.

Квалификатор restrict

`restrict`-квалифицированный указатель используется для поддержки оптимизации. Объекты, к которым обращаются косвенно через указатели, зачастую не удается оптимизировать должным образом из-за того, что на один и тот же объект теоретически могут ссылаться сразу несколько указателей. Это может затруднить оптимизацию, поскольку компилятор не знает, может ли модификация одного объекта привести к частичному изменению другого, который с виду никак с ним не связан.

Следующая функция копирует `n` байтов с участка памяти, на который указывает `q`, на участок, на который указывает `p`. Оба параметра функции, `p` и `q`, являются `restrict`-квалифицированными указателями:

```
void f(unsigned int n, int * restrict p, int * restrict q) {
  while (n-- > 0) {
    *p++ = *q++;
  }
}
```

Поскольку `p` и `q` являются `restrict`-квалифицированными указателями, компилятор может исходить из того, что обращение к объекту через один из этих параметров не приведет к обращению через другой. Это предположение основано исключительно на объявлениях параметров и не требует анализа тела функции. Использование указателей с квалификатором `restrict` делает код более эффективным, но, чтобы избежать неопределенного поведения, программист обязан обеспечить, чтобы участки памяти, на которые они ссылаются, не пересекались.

Упражнения

Попробуйте самостоятельно выполнить следующие упражнения.

1. Добавьте функцию `retrieve` в код счетчика из листинга 2.6, чтобы извлечь текущее значение `counter`.
2. Объявите массив с тремя указателями на функции и вызовите функцию, которая соответствует индексу, переданному в качестве аргумента.

Резюме

В этой главе вы познакомились с объектами и функциями и узнали, чем они отличаются. Вы научились объявлять переменные и функции, получать адреса объектов и разыменовывать объектные указатели. Вдобавок мы прошлись по большинству объектных типов, доступных программистам на C, а также по их производным типам.

Мы вернемся к ним в последующих главах, чтобы обсудить, в каких подробностях реализации их лучше всего использовать. В следующей главе вы найдете подробную информацию о двух разновидностях арифметических типов: целых числах и значениях с плавающей запятой.

3 Арифметические типы

В этой главе вы познакомитесь с двумя видами *арифметических типов*: целыми числами и значениями с плавающей запятой. Большинство операций в C поддерживают арифметические типы. Поскольку C — язык системного уровня, корректное выполнение арифметических операций может вызывать трудности и регулярно приводить к появлению дефектов. Основная причина в том, что такие операции, проводимые в цифровых системах с ограниченными диапазоном и точностью, не всегда дают математически правильные результаты. Корректное выполнение арифметических действий — фундаментальный навык любого профессионального программиста на C.

Мы подробно обсудим то, как работает арифметика в языке C, чтобы вы должным образом овладели этими основополагающими концепциями. Вы также научитесь преобразовывать одни арифметические типы в другие, что необходимо при выполнении операций со смешанными типами.

Целые числа

Как уже упоминалось в главе 2, каждый целочисленный тип представляет ограниченный диапазон значений. Целочисленные знаковые типы допускают положительные, отрицательные и нулевые значения; беззнаковые типы допускают только положительные и нулевые значения. Диапазон, охватываемый целочисленным типом, зависит от реализации.

Значение целочисленного объекта является обычным математическим числом и *представлено* в виде битов в выделенной для объекта памяти,

которые кодируют это значение. Более подробно о представлении мы поговорим позже.

Заполнение и точность

Все целочисленные типы, кроме `char`, `signed char` и `unsigned char`, могут содержать неиспользуемые биты, которые называются *заполнением* и позволяют реализации обходить причуды аппаратной платформы (такие как пропуск бита со знаком посреди представления из нескольких машинных слов) или оптимально поддерживать целевую архитектуру. Количество битов, используемых для представления значения заданного типа, не считая заполнения, но включая знак, называется *шириной* и зачастую обозначается как *N*. *Точность* — это количество битов, используемых для представления значения, не считая знака и заполнения.

Заголовочный файл <limits.h>

Заголовочный файл `<limits.h>` предоставляет минимальные и максимальные допустимые значения для различных целочисленных типов. *Допустимым* называют значение, которое можно представить с помощью того количества битов, которое доступно объекту конкретного типа. Компилятор, встречая значение, которое нельзя представить, может либо выдать диагностическое сообщение, либо преобразовать значение в представимое (хоть и некорректное). Разработчики компиляторов предоставляют для своих реализаций подходящие минимальные и максимальные значения, а также корректную ширину. Написание переносимого кода требует использования этих констант вместо целочисленных литералов наподобие `+2147483647`, которые представляют специфичные для конкретной реализации ограничения и могут измениться при переходе на другую реализацию.

Стандарт C предъявляет к размерам целых чисел всего три требования. Первое: область хранения *любого* типа данных должна иметь объем, кратный размеру объекта `unsigned char` (возможно, включая заполнение). Второе: каждый целочисленный тип должен поддерживать минимальный переносимый диапазон значений, на который можно положиться в любой реализации. Третье: меньший тип не может быть шире большего. Поэтому `USHRT_MAX`, к примеру, не может быть больше `UINT_MAX`, хотя типы `unsinged short` и `unsinged int` могут иметь одинаковую ширину.

Объявление целочисленных переменных

Целочисленный тип по умолчанию имеет знак, если только явно не объявлен как `unsigned` (исключение составляет тип `char`, который в зависимости от реализации может иметь знак, а может и не иметь). Ниже показаны корректные объявления беззнаковых целых:

```
unsigned int ui;         // unsigned требуется
unsigned u;              // можно опустить
unsigned long long ull2; // можно опустить
unsigned char uc;        // unsigned требуется
```

При объявлении целочисленных типов со знаком `signed` можно опустить; это не распространяется на тип `signed char`, который можно отличить от обычного `char` только по этому ключевому слову. Тип `int` тоже можно не указывать, если это не единственное ключевое слово. Например, переменную типа `signed long long int` нередко объявляют как `long long`, что позволяет не набирать лишний текст. Все представленные ниже объявления целочисленных переменных являются корректными:

```
int i;             // signed можно опустить
long long int sll; // signed можно опустить
long long sll2;    // signed и int можно опустить
signed char sc;    // signed требуется
```

Беззнаковые целые

Диапазон *беззнаковых целочисленных типов* начинается с 0, а их максимальное значение больше, чем у соответствующих типов со знаком. Беззнаковые целочисленные значения часто используются для представления больших неотрицательных количеств.

Представление

Беззнаковые целочисленные типы более понятны и просты в использовании, чем их аналоги со знаком. Значения в них представлены с помощью сугубо двоичной системы, без смещения: самый младший бит имеет вес 2^0, следующий за ним — 2^1 и т. д. Значение двоичного числа — это сумма весов всех битов. В табл. 3.1 показаны некоторые примеры беззнаковых значений в восьмибитном представлении без заполнения.

Таблица 3.1. Восьмибитные беззнаковые значения

Десятичное	Двоичное	Шестнадцатеричное
0	0000 0000	0x00
1	0000 0001	0x01
17	0001 0001	0x11
255	1111 1111	0xFF

Беззнаковым целочисленным типам не нужно представлять знак числа, поэтому обычно их точность на один бит выше, чем у соответствующих типов со знаком. Беззнаковые целочисленные значения находятся в диапазоне от 0 до какого-то максимального числа, которое зависит от ширины типа. Максимальное значение равно $2^N - 1$, где N — ширина. Например, в большинстве архитектур x86 используются 32-битные целые числа без заполняющих битов, поэтому объект типа `unsigned int` имеет диапазон от 0 до $2^{32} - 1$ (4 294 967 295). Константное выражение `UINT_MAX` из заголовочного файла `<limits.h>` определяет верхнюю границу этого типа для конкретной реализации. В табл. 3.2 перечислены константные выражения из `<limits.h>` для каждого беззнакового типа, минимальный диапазон, который требуется согласно стандарту, и фактический диапазон в современных реализациях x86.

Таблица 3.2. Диапазоны беззнаковых целых чисел

Константное выражение	Максимальные величины	Значение в x86	Максимальное значение для объекта типа
UCHAR_MAX	255 // $2^8 - 1$	То же	unsigned char
USHRT_MAX	65 535 // $2^{16} - 1$	То же	unsigned short int
UINT_MAX	65 535 // $2^{16} - 1$	4 294 967 295	unsigned int
ULONG_MAX	4 294 967 295 // $2^{32} - 1$	То же	unsigned long int
ULLONG_MAX	18 446 744 073 709 551 615 // $2^{64} - 1$	То же	unsigned long long int

Циклический перенос

Циклический перенос (wraparound) происходит при выполнении арифметической операции, результат которой слишком маленький (меньше 0) или большой (больше $2^N - 1$), чтобы его можно было представить в виде конкретного беззнакового целочисленного типа. В этом случае берется остаток от деления значения на N, которое на единицу больше максималь-

но допустимого значения итогового типа. В языке C циклический перенос является определенным поведением. Вызван ли он дефектом в вашем коде, зависит от контекста. Если вы что-то подсчитываете и значение переносится, то это, вероятно, ошибка. Однако в некоторых алгоритмах шифрования циклический перенос используется намеренно.

Например, код в листинге 3.1 присваивает переменной `ui` максимально допустимое значение и затем инкрементирует ее. Полученный результат нельзя представить как `unsigned int`, поэтому он циклически переносится и превращается в 0. Если это значение декрементировать, то оно снова выйдет за рамки допустимого диапазона и в результате циклического переноса превратится в `UINT_MAX`.

Листинг 3.1. Циклический перенос беззнакового целого значения

```
unsigned int ui = UINT_MAX; // 4,294,967,295 в x86
ui++;
printf("ui = %u\n", ui);    // ui равно 0
ui--;
printf("ui = %u\n", ui);    // ui равно 4,294,967,295
```

Ввиду циклического переноса беззнаковое целочисленное выражение никогда не может быть меньше 0. Об этом можно легко забыть и реализовать сравнения, которые всегда остаются истинными. Например, переменная `i` в следующем цикле `for` не может иметь отрицательного значения, поэтому данный цикл никогда не прервется:

```
for (unsigned int i = n; i >= 0; --i)
```

Это поведение стало причиной некоторых известных программных дефектов в реальных системах. Например, в Boeing 787 за каждый из шести генераторов отвечает отдельный блок управления. Если верить Федеральному управлению гражданской авиации США, то в ходе лабораторных исследований специалисты Boeing обнаружили, что внутренний программный счетчик в этом блоке выполняет циклический перенос после непрерывной работы на протяжении 248 дней[1]. В результате все шесть блоков управления генераторами, установленными на двигателях, одновременно переключаются в безопасный режим.

Чтобы избежать незапланированного поведения (такого как крушение самолета), необходимо удостовериться в отсутствии циклического

[1] См. Airworthiness Directives; The Boeing Company Airplanes, https://www.federalregister.gov/d/2015-10066/.

переноса с учетом лимитов, указанных в заголовочном файле <limits.h>. Но будьте осторожны при реализации этих проверок, поскольку в них легко ошибиться. Например, следующий код содержит дефект, так как sum + ui никогда не может быть больше, чем UINT_MAX:

```
extern unsigned int ui, sum;
// присваиваем значения переменным ui и sum
if (sum + ui > UINT_MAX)
  too_big();
else
  sum = sum + ui;
```

Если результат sum + ui больше UINT_MAX, то приводится по модулю к UINT_MAX + 1. Таким образом вся эта проверка оказывается бесполезной и сгенерированный код всегда будет выполнять сложение. Некоторые компиляторы могут выдать указывающее на это предупреждение, но это делают не все. Чтобы исправить положение, мы можем вычесть sum из обеих сторон неравенства с целью создать следующую эффективную проверку:

```
extern unsigned int ui, sum;
// присваиваем значения переменным ui и sum
if (ui > UINT_MAX - sum)
  too_big();
else
  sum = sum + ui;
```

Значение UINT_MAX — самое большое, какое может быть представлено как unsigned int, а sum находится где-то между 0 и UINT_MAX. Если sum равно UINT_MAX, то разность будет равна 0; если sum равно 0, то результат будет равен UINT_MAX. Поскольку значение, возвращаемое этой операцией, всегда находится в допустимом диапазоне (от 0 до UINT_MAX), с ним не может произойти циклический перенос.

Та же проблема возникает при сравнении результата арифметической операции с 0 (минимальным беззнаковым значением):

```
extern unsigned int i, j;
// присваиваем значения переменным i и j
if (i - j < 0) // это невозможно
  negative();
else
  i = i - j;
```

Поскольку беззнаковые целочисленные значения никогда не могут быть отрицательными, вычитание будет выполняться в любом случае. Качественные компиляторы могут предупредить и о данной ошибке. Вместо

этой бесполезной проверки мы можем сравнить `j` с `i`, чтобы узнать происходит ли циклический перенос:

```
if (j > i) // правильно
  negative();
else
  i = i - j;
```

Если `j > i`, то мы можем с уверенностью сказать, что результат будет циклически перенесен. Убрав из проверки операцию вычитания, мы исключили возможность возникновения циклического переноса во время ее выполнения.

ВНИМАНИЕ

Обратите внимание: ширина, используемая при переносе, зависит от реализации. Это значит, что на разных платформах можно получить различные результаты. Если этого не учитывать, то ваш код не будет переносимым.

Знаковые целые

У каждого беззнакового целочисленного типа (за исключением `_Bool`) есть знаковый аналог того же размера. Целочисленные типы со знаком используются для представления отрицательных, нулевых и положительных значений, диапазон которых зависит от того, сколько битов выделено для этого типа и его представления.

Представление

Целочисленные типы со знаком имеют более сложное представление, чем их беззнаковые аналоги. Язык C традиционно поддерживает три варианта представления таких значений:

- *прямой код* — старший разряд обозначает знак, а остальные разряды представляют величину значения в обычной двоичной системе;
- *обратный код* — разряду со знаком назначается вес $-(2^{N-1}-1)$, а остальные разряды значения имеют те же веса, что и в беззнаковом типе;
- *дополнительный код* — разряду со знаком назначается вес $-(2^{N-1})$, а остальные разряды значения имеют те же веса, что и в беззнаковом типе.

Вы не можете выбрать конкретное представление; оно определяется теми, кто реализует язык C для различных систем. В употреблении находятся

все три варианта, но последний намного популярнее — настолько, что комитет во главе стандарта C собирается оставить в спецификации C2x лишь дополнительный код. Далее в книге используется именно это представление.

Целочисленные типы со знаком и шириной N способны представить любое целое значение в диапазоне от -2^{N-1} до $2^{N-1} - 1$. Это, к примеру, означает, что восьмибитное значение типа `signed char` имеет диапазон от −128 до 127. Дополнительный код может также представить еще одно *самое маленькое отрицательное* значение. Для восьмибитного типа `signed char` это −128, и его модуль, |−128|, не может быть представлен данным типом. Подобное положение приводит к возникновению ряда интересных граничных ситуаций, которые мы рассмотрим позже в этой и следующей главах.

В табл. 3.3 перечислены константные выражения из заголовочного файла `<limits.h>` для каждого типа со знаком, минимальный диапазон, который требуется согласно стандарту, и фактический диапазон в современных реализациях x86.

Таблица 3.3. Диапазоны целых чисел со знаком

Константное выражение	Минимальные величины	Значение в x86	Тип
SCHAR_MIN	−127 // $-(2^7 - 1)$	−128	signed char
SCHAR_MAX	+127 // $2^7 - 1$	То же	signed char
SHRT_MIN	−32 767 // $-(2^{15} - 1)$	−32 768	short int
SHRT_MAX	+32 767 // $2^{15} - 1$	То же	short int
INT_MIN	−32 767 // $-(2^{15} - 1)$	−2 147 483 648	int
INT_MAX	+32 767 // $2^{15} - 1$	+2 147 483 647	int
LONG_MIN	−2 147 483 647 // $-(2^{31} - 1)$	−2 147 483 648	long int
LONG_MAX	+2 147 483 647 // $2^{31} - 1$	То же	long int
LLONG_MIN	−9 223 372 036 854 775 807 // $-(2^{63} - 1)$	−9 223 372 036 854 775 808	long long int
LLONG_MAX	+9 223 372 036 854 775 807 // $2^{63} - 1$	То же	long long int

Представление отрицательных чисел в дополнительном коде состоит из разрядов для хранения знака и самого значения. Разряду со знаком назначается вес $-(2^{N-1})$. Чтобы сделать значение отрицательным в до-

полнительном коде, достаточно инвертировать каждый значащий бит и затем прибавить 1 (с переносом, если это необходимо), как показано на рис. 3.1.

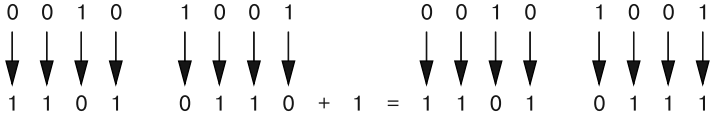

Рис. 3.1. Смена знака восьмибитного значения, представленного в дополнительном коде

В табл. 3.4 показаны двоичные и десятичные представления для восьмибитного целочисленного типа со знаком в дополнительном коде без заполнения (то есть $N = 8$). Без этой информации можно обойтись, но вам как программисту на C она скорее всего пригодится.

Таблица 3.4. Восьмибитные значения в дополнительном коде

Двоичное	Десятичное	Вес	Константа
00000000	0	0	
00000001	1	20	
01111110	126	$2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1$	
01111111	127	$2^{N-1} - 1$	SCHAR_MAX
10000000	–128	$-(2^{N-1}) + 0$	SCHAR_MIN
10000001	–127	$-(2^{N-1}) + 1$	
11111110	–2	$-(2^{N-1}) + 126$	
11111111	–1	$-(2^{N-1}) + 127$	

Переполнение

Переполнение происходит, когда операция со знаковым целым возвращает значение, которое не может быть представлено итоговым типом. Например, следующая реализация функционального макроса, которая возвращает модуль целочисленного значения, может переполниться:

```
// не определено или неверно для самого отрицательного значения
#define Abs(i) ((i) < 0 ? -(i) : (i))
```

Макросы будут подробно рассмотрены в главе 9. Пока можете считать функциональные макросы функциями, которые работают с обобщенными

типами. На первый взгляд этот макрос правильно реализует вычисление числа по модулю, возвращая неотрицательное значение `i` вне зависимости от его знака. Мы используем условную операцию (`?:`; более подробно о ней поговорим в следующей главе), чтобы проверить, является ли значение `i` отрицательным. Если оно меньше нуля, то мы меняем знак, `-(i)`; в противном случае оно возвращается без изменений `(i)`.

Мы реализовали операцию `Abs` в виде функционального макроса, поэтому она может принимать аргументы любых типов. Конечно, применять ее к беззнаковым целым числам бессмысленно, поскольку они никогда не могут быть отрицательными и макрос просто воспроизвел бы переданный ему аргумент. Но данную операцию можно использовать для различных целочисленных и вещественных типов со знаком, как в следующем примере:

```
signed int si = -25;
signed int abs_si = Abs(si);
printf("%d\n", abs_si); // выводит 25
```

Здесь мы передаем макросу `Abs` объект типа `signed int` со значением –25. Это выражение разворачивается следующим образом:

```
signed int si = -25;
signed int abs_si = ((si) < 0 ? -(si) : (si));
printf("%d\n", abs_si); // выводит 25
```

Макрос корректно возвращает модуль числа –25. Пока все хорошо. Но проблема в том, что результат смены знака для самого маленького отрицательного значения заданного типа в дополнительном коде не может быть представлен этим типом, поэтому такое применение функции `Abs` приводит к переполнению знакового целого. Следовательно, данная реализация `Abs` является дефектной и может сделать что угодно, в том числе неожиданно вернуть отрицательное значение:

```
signed int si = INT_MIN;
signed int abs_si = Abs(si); // неопределенное поведение
printf("%d\n", abs_si);
```

Что же должен вернуть вызов `Abs(INT_MIN)`, чтобы его поведение было корректным? С точки зрения языка C переполнение целочисленных типов со знаком является неопределенным поведением, что позволяет реализациям молча выполнять *циклический* перенос (происходит чаще всего), прерывание или и то и другое. *Прерывания* останавливают программу, не давая выполнить последующие операции. В распространенных архитектурах, таких как x86, используется сочетание циклического переноса и прерывания. Поскольку это неопределенное поведение, у данной

проблемы нет какого-то единого общепринятого решения. Но мы можем как минимум проверить возможность переполнения до того, как оно случится, и принять соответствующие меры.

Чтобы этот макрос мог возвращать модуль для разных типов, мы создадим для него еще один аргумент, `flag`, который зависит от типа. Данный флаг представляет макрос `*_MIN`, который соответствует типу первого аргумента, и возвращается в случае возникновения проблемы:

```
#define AbsM(i, flag) ((i) >= 0 ? (i) : ((i)==(flag) ? (flag) : -(i)))
signed int si = -25; // попробуйте INT_MIN, чтобы спровоцировать
                     // проблемный случай
signed int abs_si = AbsM(si, INT_MIN);
if (abs_si == INT_MIN)
  goto recover; // особый случай
else
  printf("%d\n", abs_si); // выводит 25
```

Макрос `AbsM` проверяет наличие самого маленького отрицательного значения и, если оно обнаружено, просто возвращает его, не пытаясь его инвертировать. Это позволяет избежать неопределенного поведения.

В некоторых системах стандартная библиотека C предоставляет следующую функцию для получения значений по модулю (только для `int`); если передать ей `INT_MIN` в качестве аргумента, то переполнение не произойдет:

```
int abs(int i) {
  return (i >= 0) ? i : -(unsigned)i; // избегает переполнения
}
```

В этом случае `i` приводится к `unsigned int` и инвертируется. Более подробно о приведении типов мы поговорим позже в данной главе.

Возможно, вас это удивит, но унарная операция вычитания (`-`) определена и для беззнаковых целочисленных типов. Результат приводится по модулю к числу, на единицу превышающему самое большое значение, которое может быть представлено итоговым типом. Наконец, `i` автоматически приводится обратно к `signed int`, как того требует оператор `return`. Поскольку `-INT_MIN` нельзя представить в виде `signed int`, результат зависит от реализации. Вот почему этот подход используется только в *некоторых системах*, и даже в них функция `abs` возвращает неправильное значение.

Функциональные макросы `Abs` и `AbsM` вычисляют свои параметры несколько раз. Это может привести к неприятным сюрпризам, если аргументы меняют состояние программы. Данное явление называется побочными эффектами, и более подробно о них мы поговорим в следующей главе.

С другой стороны, при вызовах функций каждый аргумент вычисляется только один раз.

Циклический перенос беззнаковых целых — это определенное поведение. Переполнение целочисленных значений со знаком или возможность такового всегда следует считать дефектом.

Целочисленные константы

Целочисленные константы (или *целочисленные литералы*) применяются для добавления в программу конкретных целых чисел. Например, их можно использовать в объявлении или присваивании, чтобы назначить счетчику начальное значение 0. Язык C поддерживает три вида целочисленных констант, которые основаны на разных системах счисления: десятичные, восьмеричные и шестнадцатеричные.

Десятичные константы всегда начинаются с ненулевой цифры. Например, в следующем коде используются две десятичные константы:

```
unsigned int ui = 71;
int si;
si = -12;
```

В этом примере мы инициализируем `ui` с помощью десятичной константы `71` и присваиваем `si` десятичное константное значение `-12`. Используйте десятичные константы, когда вам нужно добавить в свой код обычные целые числа.

Если литерал начинается с 0, вслед за которым могут идти цифры от 0 до 7, то это *восьмеричная константа*. Ниже представлен пример:

```
int agent = 007;
int permissions = 0777;
```

В этом коде восьмеричный литерал `007` эквивалентен десятичному литералу 7, а восьмеричная константа `0777` равна десятичному значению 511. С помощью восьмеричных констант, к примеру, удобно работать с трехбитными полями.

Вы также можете создавать *шестнадцатеричные константы*, указав `0x` или `0X` в начале последовательности из десятичных цифр и букв от a (или A) до f (или F). Например:

```
int burger = 0xDEADBEEF;
```

Используйте шестнадцатеричные константы, когда вы хотите представить последовательность битов, а не обычное число, например при представлении адреса. Принято, что большинство шестнадцатеричных констант записываются как `0xDEADBEEF`, поскольку это напоминает типичный дамп памяти в шестнадцатеричном формате. Вам, пожалуй, стоит записывать все свои шестнадцатеричные константы именно так.

К константе также можно добавить суффикс, чтобы указать ее тип. Без этого десятичной константе назначается тип `int`, если она может быть представлена значением данного типа; если нет, то используется тип `long int` или `long long int`. Поддерживаются следующие суффиксы: `U` для `unsigned`, `L` для `signed long` и `LL` для `long long`. Их можно объединять. Например, суффикс `ULL` представляет тип `unsigned long long`. Ниже представлены некоторые примеры:

```
unsigned int ui = 71U;
signed long int sli = 9223372036854775807L;
unsigned long long int ui = 18446744073709551615ULL;
```

Если не использовать суффикс и целочисленная константа не имеет подходящего типа, то она может быть автоматически преобразована (об автоматических преобразованиях мы поговорим в разделе «Арифметическое преобразование» на с. 83). Это может привести к неожиданным преобразованиям или диагностическим сообщениям компилятора, и потому для целочисленных констант лучше указывать нужный вам тип. Больше информации о целочисленных константах можно найти в разделе 6.4.4.1 стандарта C.

Числа с плавающей запятой

Числа с плавающей запятой[1] — самое распространенное представление вещественных значений в компьютерах. Это методика, которая использует экспоненциальную запись для кодирования чисел в виде мантиссы и порядка. Например, десятичное число `123.456` можно представить как $1{,}23456 \times 10^2$, а двоичное `0b10100.110` — как $1{,}0100110 \times 2^4$.

Представление с плавающей запятой можно сгенерировать несколькими способами. Стандарт C не требует от реализаций использования какой-то

[1] Поскольку в России целая часть числа от дробной традиционно отделяется запятой, то для обозначения того же понятия исторически используется термин «плавающая запятая». — *Примеч. ред.*

конкретной модели, а лишь отмечает, что *какая-то* модель должна поддерживаться. Чтобы не усложнять, будем исходить из соответствия приложению F стандарта языка C, которое описывает самый распространенный формат плавающей запятой. В современных компиляторах можно проверить значения макросов `__STDC_IEC_559__` или `__STDC_IEC_60559_BFP__`, чтобы определить, соответствует ли реализация этому формату.

В этом разделе мы рассмотрим типы, арифметические операции, значения и константы с плавающей запятой. Вы узнаете, как и когда их следует использовать для имитации работы с вещественными числами и в каких ситуациях их лучше избегать.

Типы с плавающей запятой

Язык C поддерживает три типа с плавающей запятой: `float`, `double` и `long double`.

Тип `float` можно использовать в вычислениях с плавающей запятой, в которых результат можно адекватно представить с одинарной точностью. Согласно распространенной спецификации IEC 60559 при кодировании типа `float` один разряд выделяется для знака, восемь — для порядка и еще 23 — для мантиссы (ISO/IEC/IEEE 60559:2011).

Тип `double` имеет более высокую точность, но занимает дополнительное место. При его кодировании один разряд отводится для знака, 11 — для порядка и еще 52 — для мантиссы. Эти типы показаны на рис. 3.2.

Во всех реализациях тип `long double` должен иметь какой-то из следующих форматов:

- формат числа четверной точности IEC 60559 (или binary128)[1];
- формат двойной точности IEC 60559 (расширенная версия binary64);
- расширенный формат, несовместимый с IEC 60559;
- формат двойной точности IEC 60559 (или binary64).

Разработчикам компиляторов рекомендуется использовать для типа `long double` формат IEC 60559 binary128 или расширенную версию IEC 60559 binary64. Последняя включает в себя распространенный 80-битный формат IEC 60559.

[1] В 2011 году в спецификации IEC 60559 к основным форматам был добавлен binary128.

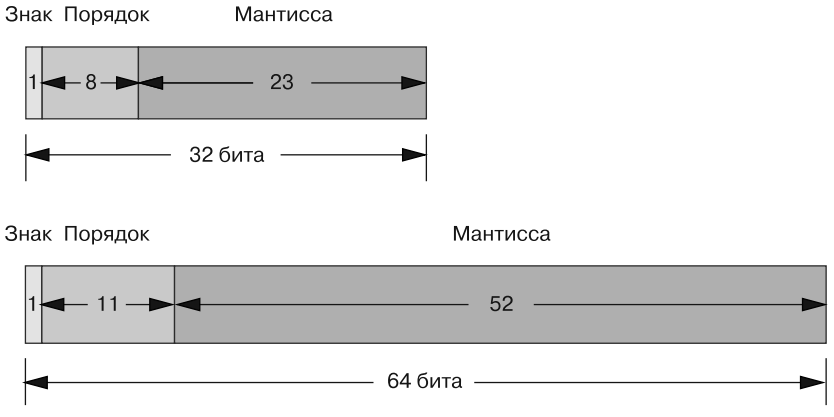

Рис. 3.2. Типы float и double

Бóльшие типы имеют повышенную точность, но занимают больше места. Любое значение, которое можно представить как `float`, может уместиться в `double`, а значение, которое можно представить как `double`, поместится в `long double`. В будущих версиях стандарта могут появиться дополнительные типы с плавающей запятой, которые, возможно, будут иметь более высокую точность или расширенный диапазон (или и то и другое) по сравнению с `long double` либо же меньшие диапазон и точность, например 16-битный тип с плавающей запятой.

Знак, порядок и мантисса

Как и в случае с целыми, знаковый бит (или *разряд*) определяет, является число положительным или отрицательным; 0 обозначает положительное число, а 1 — отрицательное.

Порядок должен представлять как положительные, так и отрицательные степени. Чтобы не хранить порядок в виде числа со знаком, для получения его *хранимой* величины к непосредственной степени прибавляется смещение. Для типа `float` смещение равно 127. Следовательно, чтобы выразить степень 0, в качестве порядка нужно сохранить 127. Хранимая величина 200 обозначает порядок 200 – 127, или 73. Порядки –127 (когда каждый разряд порядка равен 0) и +128 (когда каждый разряд порядка равен 1) зарезервированы для особых чисел. Числа двойной точности тоже имеют смещение, 1023. Это значит, для `float` хранимое значение находится в диапазоне от 0 до 255, а для `double` — в диапазоне от 0 до 2047.

Разряды, относящиеся к *мантиссе*, определяют точность числа. Например, если представить $1{,}0100110 \times 2^4$ в виде значения с плавающей запятой, то 1,0100110 будет мантиссой, а степень 2 (которая равна 4) — порядком (Холлаш, 2019).

Арифметические операции с плавающей запятой

Числа с плавающей запятой похожи на вещественные числа и когда-то использовались для их моделирования. Но между ними есть важные различия. В частности, числа с плавающей запятой, в отличие от вещественных, ограничены по величине и точности. Операции сложения и умножения *не* являются ассоциативными, распределительный закон *не* выполняется. То же самое относится и к другим свойствам, которыми обладают вещественные числа.

Типы с плавающей запятой не могут точно представить все вещественные значения, даже если те состоят из небольшого количества десятичных цифр. Например, распространенные десятичные константы наподобие 0,1 не могут быть точно представлены в виде двоичных чисел с плавающей запятой. Типам с плавающей запятой может не хватать точности для применения в качестве счетчиков в циклах или в финансовых расчетах. См. правило FLP30-C из стандарта программирования CERT C (не используйте переменные с плавающей запятой в качестве счетчиков в циклах).

Значения с плавающей запятой

Обычно все разряды мантиссы в типе с плавающей запятой выражают значимые цифры (в том числе *ведущая* единица, которая опускается, но все равно считается частью значения). Число 0 является особым случаем, и для его представления порядок и мантисса должны быть равны 0; знак нуля (+0 или –0) определяется соответствующим разрядом, поэтому существует два нулевых значения с плавающей запятой: положительное и отрицательное.

Мантисса нормального значения с плавающей запятой не начинается с нулей; ведущие нули убираются за счет изменения порядка. Таким образом, `float` имеет мантиссу с 24-битной точностью, `double` — с 53-битной, а `long double` — с 113-битной (при условии соблюдения 128-битного формата IEC 60559). Это *нормализованные* числа, которые сохраняют полную точность мантиссы.

Существуют также денормализованные (или субнормальные) числа. Они имеют очень маленькие положительные или отрицательные величины (но не 0), представление которых имело бы порядок меньше допустимого. На рис. 3.3 изображен диапазон ненормализованных значений по обе

стороны от 0. Ненулевое число, представленное минимально допустимым порядком (то есть подразумевается, что опущенный разряд 1 равен 0), является ненормализованным, даже если все явно обозначенные разряды мантиссы равны 1. По своей точности денормализованные значения с плавающей запятой уступают нормализованным.

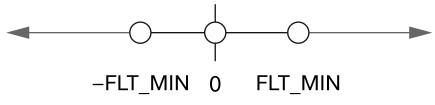

Рис. 3.3. Диапазон ненормализованных чисел

Типы с плавающей запятой также могут представлять значения, которые не являются числами с плавающей запятой, — такие как положительная и отрицательная бесконечность и NaN (not-a-number, не число). NaN — это значения, которые не соответствуют настоящим числам.

Возможность представить бесконечность в виде конкретного значения позволяет не останавливаться на переполнении; это зачастую дает желаемый результат и не требует выполнения особых действий. Например, при делении любого положительного или отрицательного ненулевого числа на положительный или отрицательный нуль[1] получается либо плюс, либо минус бесконечность. Операции с бесконечными значениями четко определены в стандарте IEEE для чисел с плавающей запятой.

Тихое NaN может пройти почти через любую арифметическую операцию, не сгенерировав исключения с плавающей запятой, и обычно проверяется после выбранной последовательности операций. *Сигнальное NaN* обычно генерирует исключение с плавающей запятой сразу же, как только его используют в качестве арифметического операнда. Эти исключения являются сложной темой, которая здесь не рассматривается. Более подробную информацию о них можно прочесть в приложении F стандарта C.

В спецификации IEC 60559 значения NaN и бесконечности обозначаются с помощью макросов `NAN` и `INFINITY`, а также функции `nan` из заголовочного файла `<math.h>`. Макросы `SNANF`, `SNAN` и `SNANL` (ISO/IEC TS 18661-1:2014,

[1] –0 и +0 являются отдельными значениями, которые при сравнении оказываются равными.

ISO/IEC TS 18661-3:2015), определенные в заголовочном файле `<math.h>`, предоставляют обозначения для сигнальных NaN. В стандарте C полная поддержка последних не требуется.

Чтобы определить, с какого рода значением с плавающей запятой вы имеете дело, можно воспользоваться функциональным макросом `fpclassify`, который классифицирует переданный ему аргумент как NaN, бесконечность, нормализованный, денормализованный или нуль.

```
#include <math.h>
int fpclassify(real-floating x);
```

В листинге 3.2 макрос `fpclassify` используется в функции `show_classification` для определения того, что собой представляет значение с плавающей запятой типа `double`: нормализованное число, ненормализованное число, ноль, бесконечность или NaN.

Листинг 3.2. Макрос fpclassify

```
const char *show_classification(double x) {
  switch(fpclassify(x)) {
    case FP_INFINITE:  return "Inf";
    case FP_NAN:       return "NaN";
    case FP_NORMAL:    return "normal";
    case FP_SUBNORMAL: return "subnormal";
    case FP_ZERO:      return "zero";
    default:           return "unknown";
  }
```

Аргумент функции `x` (в данном случае `double`) передается макросу `fpclassify`, который возвращает результат в выражение `switch`. Функция возвращает строку, соответствующую классу значения, которое хранится в `x`.

Константы с плавающей запятой

Константа с плавающей запятой — это десятичное или шестнадцатеричное число, представляющее вещественное значение со знаком. Такие константы следует использовать для хранения значений с плавающей запятой, которые нельзя изменить. Ниже приведено несколько примеров констант с плавающей запятой:

```
15.75
1.575E1   /* 15.75 */
```

```
1575e-2   /* 15.75 */
-2.5e-3   /* -0.0025 */
25E-4     /* 0.0025 */
```

У всех констант с плавающей запятой есть тип. При отсутствии суффикса используется тип `double`; тип `float` имеет суффикс `f` или `F`, а суффиксы `l` и `L` предназначены для типа `long double`, как показано ниже:

```
10.0    /* тип double */
10.0F   /* тип float */
10.0L   /* тип long double */
```

Арифметическое преобразование

Значения, имеющие некий определенный тип (например, `float`), часто приходится представлять в других типах (таких как `int`). Подобная необходимость может возникнуть, когда у вас, скажем, есть объект типа `float` и вы хотите передать его в качестве аргумента функции, которая принимает только объекты типа `int`. При выполнении таких преобразований обязательно нужно убедиться в том, что значение может быть представлено новым типом. Мы обсудим это более подробно в подразделе «Безопасное приведение типов» на с. 89.

Значения одного типа могут приводиться к другому явно и неявно. В первом случае можно использовать операцию *приведения типов*. Соответствующие примеры показаны в листинге 3.3.

Листинг 3.3. Операции приведения типов

```
int si = 5;
short ss = 8;
long sl = (long)si; ❶
unsigned short us = (unsigned short)(ss + sl); ❷
```

Чтобы выполнить преобразование, укажите непосредственно перед выражением имя типа в круглых скобках. Это приведет выражение к неквалифицированной версии заданного типа. Здесь мы приводим значение `si` к типу `long` ❶. Поскольку `si` имеет тип `int`, данное преобразование (от меньшего целочисленного типа к большему без добавления или удаления знака) является гарантированно безопасным, поскольку значение всегда можно представить в большем типе.

Второе преобразование в этом фрагменте кода ❷ приводит результат выражения `(ss + sl)` к типу `unsigned short`. Поскольку тип `(unsigned short)` имеет меньшую точность, результат преобразования может не совпасть с исходным значением (одни компиляторы об этом предупреждают, а другие — нет). В данном примере результат выражения (13) может быть корректно представлен в итоговом типе.

Неявное преобразование происходит автоматически в выражениях по мере необходимости — например, при выполнении операций со смешанными типами. В представленном выше листинге 3.3 неявные преобразования приводят `ss` к типу переменной `s1`, чтобы сложение `ss + sl` выполнялось в рамках одного типа. Правила относительно неявного приведения значений к тем или иным типам довольно сложны и основаны на трех концепциях: ранге преобразования, повышении разрядности целочисленных значений и обычных арифметических преобразованиях. Обсудим их в следующих подразделах.

Ранг преобразования целочисленных типов

У каждого целочисленного типа есть *ранг преобразования* — порядковый номер, который определяет, как и при каких условиях происходит неявное приведение типа.

Пункт 1 подраздела 6.3.1.1 стандарта C (ISO/IEC 9899:2018) гласит, что каждый целочисленный тип имеет ранг преобразования, на который распространяются следующие правила:

- все целочисленные типы со знаком имеют разный ранг, даже если у них одинаковое представление;
- целочисленный тип со знаком всегда имеет больший ранг, чем любой беззнаковый целочисленный тип с меньшей точностью;
- у типа `long long int` более высокий ранг, чем у типа `long int`; у `long int` более высокий ранг, чем у `int`; у `int` более высокий ранг, чем у `short int`; у `short int` более высокий ранг, чем у `signed char`;
- ранг любого беззнакового целочисленного типа равен рангу соответствующего целочисленного типа со знаком, если таковой имеется;
- тип `char` имеет тот же ранг, что `signed char` и `unsigned char`;

- тип `_Bool` имеет меньший ранг, чем любой другой стандартный целочисленный тип;
- ранг любого перечисляемого типа равен рангу совместимого целочисленного типа. Любой перечисляемый тип совместим с `signed int` и `unsinged int`;
- относительные ранги преобразования любых расширенных целочисленных типов со знаком, имеющих одинаковую точность, определяются на уровне реализации, но при этом должны соблюдать все остальные правила порядка приведения целочисленных типов.

Повышение разрядности целочисленных значений

Малый тип — целочисленное значение, ранг преобразования которого ниже, чем у `int` или `unsigned int`. Повышение разрядности — это процесс приведения значений малых типов к `int` или `unsigned int`. Он позволяет использовать малые типы в любых выражениях, в которых могут применяться `int` и `unsigned int`. Например, вы можете использовать целочисленный тип с меньшим рангом (обычно `char` или `short`) по правую сторону операции присваивания или в качестве аргумента функции.

Повышение разрядности служит двум основным целям. Во-первых, оно поощряет проведение операций с использованием естественного для архитектуры размера, что улучшает производительность. Во-вторых, помогает избежать арифметических ошибок, которые возникают из-за переполнения промежуточных значений. Это показано в следующем фрагменте кода:

```
signed char cresult, c1, c2, c3;
c1 = 100; c2 = 3; c3 = 4;
cresult = c1 * c2 / c3;
```

Без повышения разрядности выражение `c1 * c2` привело бы к переполнению типа `signed char` на тех платформах, где он представлен восьмибитным значением в дополнительном коде, так как 300 выходит за пределы диапазона значений, которые могут быть представлены объектом этого типа (от –128 до 127). Но благодаря повышению разрядности `c1`, `c2` и `c3` автоматически преобразуются в объекты типа `signed int`, и операции умножения и деления выполняются в данном размере. При этом не может

произойти переполнение, поскольку итоговые значения всегда могут быть представлены типом `signed int` (объекты которого имеют диапазон от -2^{N-1} до $2^{N-1}-1$). В этом конкретном примере результат всего выражения равен 75; он находится в диапазоне типа `signed char`, поэтому, когда он присваивается переменной `cresult`, его значение сохраняется.

До появления первого стандарта C компиляторы выполняли повышение разрядности одним из двух способов: *с сохранением беззнаковости* и *с сохранением значения*. В первом случае компилятор расширяет малый беззнаковый тип до `unsigned int`. Во втором значение исходного малого типа приводится к `int`, если тот способен представить все его значения; если нет, то значение приводится к `unsigned int`. В ходе работы над первой версией стандарта (C89) было решено остановиться на подходе с сохранением значения, поскольку он реже приводит к некорректным результатам, чем его альтернатива. При необходимости вы можете переопределить данное поведение, воспользовавшись ручным приведением типов, как это было сделано в листинге 3.3.

Результат повышения разрядности малых беззнаковых типов зависит от точности целочисленных значений, которая определяется реализацией. Например, архитектура x86 поддерживает восьмибитный тип `char`, 16-битный тип `short` и 32-битный тип `int`. В реализациях, рассчитанных на эту архитектуру, малые типы `unsigned char` и `unsigned short` расширяются до `signed int`, поскольку `signed int` может представить любые их значения. Однако в 16-битных архитектурах, таких как Intel 8086/8088 и IBM Series/1, тип `char` занимает восемь бит, а `short` и `int` — по 16. В реализациях, рассчитанных на эти архитектуры, значения типа `unsigned char` расширяются до `signed int`, тогда как значения типа `unsigned short` расширяются до `unsigned int`. Это обусловлено тем, что все значения восьмибитного типа `unsigned char` могут быть представлены 16-битным типом `signed int`, но некоторые значения 16-битной версии `unsigned short` не помещаются в 16-битный тип `signed int`.

Обычные арифметические преобразования

Обычные арифметические преобразования — это правила выбора общего типа за счет балансирования обоих операндов бинарной операции или путем балансирования второго и третьего аргументов условной операции (`? :`).

Балансирующее преобразование приводит один или оба операнда, имеющие разные типы, к одному и тому же типу. Большинство операций, которые принимают целочисленные операнды, включая `*`, `/`, `%`, `+`, `-`, `<`, `>`, `<=`, `>=`, `==`, `!=` , `&`, `^`, `|` и `? :`, совершают обычные арифметические преобразования. Эти преобразования применяются к операндам с повышенной разрядностью.

Обычные арифметические преобразования сначала проверяют, имеет ли один из операндов тип с плавающей запятой. Если да, то действуют следующие правила.

1. Если любой из операндов имеет тип `long double`, то другой тоже приводится к `long double`.
2. Иначе если любой из операндов имеет тип `double`, то другой тоже приводится к `double`.
3. В противном случае если любой из операндов имеет тип `float`, то другой тоже приводится к `float`.
4. В противном случае повышение разрядности выполняется для обоих операндов.

Например, если один операнд имеет тип `double`, а другой `int`, то последний преобразуется в объект типа `double`. Если один операнд имеет тип `float`, а другой `double`, то первый преобразуется в объект типа `double`.

Если ни один из операндов не является значением типа с плавающей запятой, то к целочисленным операндам с повышенной разрядностью применяются следующие правила обычного арифметического преобразования.

1. Если оба операнда имеют один и тот же тип, то дальнейших преобразований не требуется.
2. В противном случае если оба операнда знаковые целые или оба беззнаковые целые, то операнд с меньшим рангом преобразования приводится к типу операнда с более высоким рангом. Например, если один операнд имеет тип `int`, а другой — `long`, то операнд типа `int` приводится к `long`.
3. В противном случае если операнд с беззнаковым целочисленным типом имеет ранг больший или равный рангу типа другого операнда, то операнд со знаком приводится к беззнаковому целочисленному типу.

Например, если один операнд имеет тип `signed int`, а другой — `unsigned int`, то операнд типа `signed int` приводится к `unsigned int`.

4. В противном случае если тип целочисленного операнда со знаком может представить все значения операнда с беззнаковым целочисленным типом, то последний приводится к целочисленному типу со знаком. Например, если один операнд имеет тип `unsigned int`, а другой — `signed long long` и если второй может представить все значения первого, то операнд типа `unsigned int` преобразуется в объект типа `signed long long`. Это происходит в реализациях с 32-битным типом `int` и 64-битным типом `long long`, таких как x86-32 и x86-64.
5. В противном случае операнды приводятся к беззнаковым версиям своих целочисленных типов со знаком.

Эти правила преобразования эволюционировали по мере добавления в ранние версии языка C новых типов, и к ним необходимо привыкнуть. Несоответствия, которые в них наблюдаются, возникают из-за разных архитектурных свойств (в частности, автоматическое расширение `char` до `int` в PDP-11), нежелания менять поведение существующих программ и стремления к единообразию (с учетом этих ограничений). Если сомневаетесь, то используйте ручное приведение типов, чтобы обеспечить нужное вам преобразование. Но старайтесь не злоупотреблять ручным приведением, поскольку оно может препятствовать важным диагностическим механизмам компилятора.

Пример автоматического приведения типов

Следующий пример иллюстрирует использование порядка приведения и повышения разрядности целочисленных типов, а также обычные арифметические преобразования. Этот код проверяет равенство между значением `c` типа `signed char` и значением `ui` типа `unsigned int`. Мы будем исходить из того, что он компилируется для архитектуры x86:

```
unsigned int ui = UINT_MAX;
signed char c = -1;
if (c == ui) {
  puts("-1 equals 4,294,967,295");
}
```

Переменная `c` имеет тип `signed char`. Поскольку `signed char` по своему рангу целочисленных преобразований уступает типам `int` и `unsigned int`,

во время сравнения значение, хранящееся в `c`, расширяется до объекта типа `signed int`. Это достигается за счет *расширения знака* исходного значения с `0xFF` до `0xFFFFFFFF`. Такое приведение используется для преобразования значений со знаком в более широкие объекты. Разряд, хранящий знак, копируется в каждый разряд расширенного объекта. Данная операция сохраняет знак и величину в ходе приведения меньшего целочисленного значения со знаком к большему.

Затем происходят обычные арифметические преобразования. Поскольку операнды операции равенства (`==`) имеют одинаковый ранг, но у одного из них есть знак, а у другого — нет, операнд со знаком приводится к беззнаковому целочисленному типу другого операнда. Затем выполняется 32-битная беззнаковая операция сравнения. Расширенное и преобразованное значение `c` совпадает с `UINT_MAX`, поэтому сравнение возвращает 1 и фрагмент кода выводит следующее:

```
-1 equals 4,294,967,295
```

Безопасное приведение типов

Автоматическое и ручное приведение типов может давать результат, который нельзя представить с помощью итогового типа. Чтобы избежать преобразований, операции желательно проводить с объектами одного типа. Но если функция возвращает или принимает объект другого типа, то без приведения не обойтись. В таких ситуациях следите за тем, чтобы преобразование выполнялось корректно.

Приведение целочисленных значений

Приведение этого вида происходит, когда значение целочисленного типа приводится к другому целочисленному типу. Приведение к типам большей разрядности без добавления или удаления знака всегда является безопасным, и его не нужно проверять. Большинство других преобразований может давать непредсказуемые результаты, если результат нельзя представить с помощью итогового типа. Для их корректного выполнения необходимо убедиться, что значение, хранящееся в исходном целочисленном типе, находится в допустимом диапазоне итогового целочисленного типа. Например, функция `do_stuff`, показанная в листинге 3.4, принимает аргумент `value` типа `signed long` и использует его в контексте, в котором

подходит только `signed char`. Чтобы безопасно выполнить это приведение, функция проверяет, можно ли представить данное значение в виде `signed char` с диапазоном [`SCHAR_MIN`, `SCHAR_MAX`], и если нет, то возвращает ошибку.

Листинг 3.4. Безопасное приведение типов

```
#include <errno.h>
#include <limits.h>

errno_t do_stuff(signed long value) {
  if ((value < SCHAR_MIN) || (value > SCHAR_MAX)) {
    return ERANGE;
  }
  signed char sc = (signed char)value; // приведение типов глушит
                                       // предупреждения
  //---snip---
}
```

Диапазон, который нужно проверять, зависит от конкретного преобразования. См. правило INT31-C из стандарта программирования CERT C (убедитесь в том, что приведение типов не приводит к потере или неправильной интерпретации данных).

Приведение целочисленных значений к типам с плавающей запятой

Если целочисленное значение, приводимое к типу с плавающей запятой, может быть точно представлено этим типом, то оно не меняется. Если приводимое значение находится в диапазоне, который может быть представлен, но не точно, то результат округляется к ближайшему большему или меньшему допустимому значению в зависимости от реализации. Если приводимое значение находится вне допустимого диапазона, то поведение не определено. В правиле FLP36-C стандарта программирования CERT C (в ходе приведения целочисленных значений к типу с плавающей запятой сохраняйте точность) можно найти больше информации и примеров таких преобразований.

Приведение значений с плавающей запятой к целочисленным типам

Когда конечное значение с плавающей запятой приводится к любому целочисленному типу (кроме `bool`), дробная часть отбрасывается. Если значение целой части нельзя представить с помощью целочисленного типа, то поведение не определено.

Уменьшение разрядности значений с плавающей запятой

Приведение чисел с плавающей запятой к более крупным вещественным типам всегда является безопасным. Уменьшение разрядности значения с плавающей запятой (то есть приведение к меньшему типу с плавающей запятой) похоже на приведение целого числа к типу с плавающей запятой.

Типы с плавающей запятой могут поддерживать бесконечность со знаком в формате приложения F. Уменьшение разрядности вещественных типов в таких реализациях всегда проходит успешно, поскольку все значения находятся в допустимом диапазоне. В правиле FLP34-C стандарта программирования CERT C (следите за тем, чтобы результаты преобразований с плавающей запятой находились в диапазоне нового типа) можно найти больше информации о приведении значений с плавающей запятой.

Резюме

В этой главе вы познакомились с целочисленными и вещественными типами. Мы также рассмотрели автоматические и ручные преобразования, порядок приведения целочисленных типов, повышение разрядности целочисленных значений и обычные арифметические преобразования.

В следующей главе речь пойдет об операциях и о написании простых выражений для выполнения операций с объектами различных типов.

4 Выражения и операции

В этой главе вы познакомитесь с операциями и узнаете, как писать простые выражения для выполнения операций с объектами различных типов. *Операции* могут обозначаться как ключевым словом, так и одним или несколькими знаками препинания и предназначены для выполнения команд. Когда их применяют к одному или нескольким операндам, операции становятся выражениями, которые вычисляют значения и могут иметь побочные эффекты. *Выражение* — это последовательность операций и операндов, которая вычисляет значение или выполняет какую-то другую задачу. Операндами могут быть идентификаторы, константы, строковые литералы и другие выражения.

В этой главе мы обсудим простое присваивание, а затем сделаем шаг назад и рассмотрим принцип работы выражений (операции и операнды, вычисление значений, побочные эффекты, приоритет и порядок выполнения). Затем перейдем к отдельным операциям, включая арифметические, битовые, условные, а также операции приведения типов, выравнивания, сравнения, составного присваивания и «запятая». Со многими из этих операций и выражений мы уже сталкивались в предыдущих главах; здесь же подробно поговорим об их поведении и увидим, как их лучше всего использовать. В завершение затронем тему арифметических операций с указателями.

Простое присваивание

Простое присваивание заменяет значение объекта, указанного в левой части операции, правым операндом. Значение правого операнда приводится к типу выражения присваивания. Простое присваивание состоит

из трех элементов: левого операнда, операции присваивания (=) и правого операнда. Это показано в следующем примере:

```
int i = 21; // объявление с инициализатором
int j = 7;  // объявление с инициализатором
i = j;      // простое присваивание
```

В первых двух строчках содержатся *объявления*, которые инициализируют `i` и `j` с помощью значений 21 и 7 соответственно. *Инициализатор* использует выражение присваивания, но сам таковым не является; он всегда является частью объявления.

В третьей строчке происходит простое присваивание. Чтобы ваш код скомпилировался, вы должны определить или объявить все идентификаторы, которые фигурируют в такого рода выражении.

В ходе простого присваивания правое значение приводится к типу левого и затем сохраняется в объект, который находится в левой части. В случае с `i = j` значение считывается из `j` и записывается в `i`. Поскольку `i` и `j` имеют один и тот же тип (`int`), преобразование не требуется. Выражение присваивания имеет значение результата операции присваивания и тип левого операнда.

Левый операнд в простом присваивании всегда является выражением (с любым объектным типом, кроме `void`), и мы называем его *l-значением* (lvalue). Изначально буква *l* означала *left* (левый операнд), но этот термин, вероятно, будет правильнее расшифровывать как *значение-локатор (locator value)*, поскольку он должен обозначать объект. В этом примере идентификаторы обоих объектов, `i` и `j`, представляют собой *l*-значения. *L*-значение также может принимать вид выражения наподобие `*(p+4)` — при условии, что оно ссылается на объект в памяти.

Правый операнд тоже является выражением, но при этом может иметь вид обычного значения, и ему не обязательно идентифицировать объект. Мы называем его *r-значением* (rvalue) (*r* от *right* — правое значение или операнд) или *значением выражения*. Как видно в следующем фрагменте кода, в котором используются типы и значения из предыдущего примера, *r*-значение может не ссылаться на объект:

```
j = i + 12; // j теперь имеет значение 19
```

Выражение `i + 12` — это не *l*-значение, поскольку в нем нет объекта для хранения результата. Переменная `i` сама по себе является *l*-значением, которое автоматически превращается в *r*-значение, чтобы его можно было задействовать как операнд в операции сложения. Результат сложения не связан

ни с каким участком памяти и тоже является *r*-значением. В языке C есть правила относительно того, где могут находиться *l*- и *r*-значения. Их корректное и некорректное использование показано в следующем примере:

```
int i;
i = 5;     // i — это l-значение, а 5 — r-значение
int j = i; // l-значения могут находиться в правой части присваивания
7 = i;     // Ошибка: r-значения не могут находиться в левой части
           // присваивания
```

Присваивание `7 = i` не сработает, поскольку *r*-значение всегда должно быть по правую сторону от операции. В следующем примере правый операнд имеет не тот тип, что у выражения присваивания, поэтому значение `i` предварительно приводится к типу `signed char`. Затем значение выражения, заключенного в скобки, приводится к типу `long int`:

```
signed char c;
int i = INT_MAX;
long k;
k = (c = i);
```

В ходе присваивания следует учитывать практические ограничения. В частности, простое присваивание может привести к усечению, если значение приводится к более узкому типу. Как уже упоминалось в главе 3, каждому объекту для хранения значения требуется фиксированное количество байтов. Значение `i` всегда можно представить с помощью `k` (большего типа, который не требует добавления или удаления знака). Однако в этом примере значение `i` приводится к `signed char` (к типу выражения присваивания `c = i`). Затем значение выражения, заключенного в скобки, приводится к типу внешнего выражения присваивания, то есть `long int`. Если предположить, что у `c` не хватает разрядов для представления значения, хранящегося в `i`, значения меньше `SCHAR_MAX` усекаются, то усечению подлежит и конечный результат, который хранится в `k` (–1). Во избежание этого убедитесь в том, что выбрали достаточно широкий тип, способный представить любое значение, которое может возникнуть, или делайте проверки на случай переполнения.

Оценивание

Итак, мы рассмотрели простое присваивание. Теперь немного отвлечемся и посмотрим на то, как на самом деле оцениваются выражения. *Оценивание (evaluation)* в основном выглядит как упрощение выражения вплоть до

какого-то одного значения. Однако, помимо вычислений, этот процесс может иметь побочные эффекты.

Расчет значения (value computation) — это его подсчет, происходящий в результате вычисления выражения. В ходе расчета итогового значения иногда необходимо идентифицировать объект или прочитать значение, которое было ему присвоено ранее. Например, следующее выражение содержит несколько расчетов, направленных на идентификацию `i`, `a` и `a[i]`:

```
a[i] + f() + 9
```

Поскольку `f` — функция, а не объект, выражение `f()` не требует идентификации `f`. Расчет значений операндов должно происходить до вычисления результата работы операции. В данном примере выполняются отдельные расчеты, которые считывают значение `a[i]` и определяют результат, возвращаемый вызовом функции `f`. Затем третья вычислительная операция суммирует полученные значения, чтобы узнать, чему равно все выражение целиком. Если `a[i]` — массив типа `int`, а `f()` возвращает `int`, то результат выражения тоже будет иметь тип `int`.

Побочные эффекты — это изменения состояния среды выполнения. К ним относятся запись в объект, обращение к объекту с квалификатором `volatile`, ввод/вывод, присваивание и вызов функции, которая выполняет любое из этих действий. Предыдущий пример можно немного модифицировать за счет добавления присваивания. Его побочным эффектом будет обновление значения, которое хранится в `j`:

```
int j;
j = a[i] + f() + 9;
```

Присваивание значения переменной `j` — это побочный эффект, который меняет состояние среды выполнения. В зависимости от определения функции `f`, ее вызов тоже может иметь побочные эффекты.

Вызов функции

Обозначение функции — это выражение функционального типа, которое используется для вызова функции. В следующем вызове обозначением функции выступает `max`:

```
int x = 11;
int y = 21;
int max_of_x_and_y = max(x, y);
```

Функция `max` принимает два аргумента и возвращает тот, который больше. Во время выполнения обозначение функции внутри выражения преобразуется в *указатель на функцию соответствующего типа*. Значение каждого аргумента должно иметь тип, который можно назначить объекту с (неквалифицированным) типом соответствующего параметра. Количество и тип аргументов должны соответствовать количеству и типу параметров, которые принимает функция. В данном случае это два целочисленных аргумента. Язык C также поддерживает *вариативные функции*, способные принимать переменное число аргументов (примером этого является функция `printf`).

Мы также можем передать одну функцию другой, как показано в листинге 4.1.

Листинг 4.1. Передача одной функции в другую

```
int f(void) {
  // ---snip---
  return 0;
}
void g(int (*func)(void)) {
  // ---snip---
  if (func() != 0)
    printf("g failed\n";
  // ---snip---
}
// ---snip---
g(f); // вызываем g с аргументом в виде указателя на функцию
// ---snip---
```

Этот код передает адрес функции, обозначенной как `f`, другой функции, `g`. Последняя принимает указатель на функцию, которая не принимает никаких аргументов и возвращает `int`. Функция, передаваемая в качестве аргумента, автоматически преобразуется в указатель на функцию. Определение `g` делает это явным образом; в качестве эквивалентного объявления можно было бы указать `void g(int func(void))`.

Операции инкремента и декремента

Операции *инкремента* (`++`) и *декремента* (`--`) соответственно увеличивают и уменьшают на 1 изменяемое *l*-значение. Оба они принимают по одному операнду, что делает их *унарными*.

Эти операции могут быть как *префиксными* (то есть находиться перед операндом), так и *постфиксными* (находиться после операнда). Префиксные и постфиксные операции ведут себя по-разному, из-за чего их часто используют в качестве каверзных вопросов в ходе собеседований и тестов. Префиксный инкремент выполняется перед возвращением значения, а постфиксный — после. Это показано в листинге 4.2, где результаты префиксных/постфиксных инкрементов/декрементов присваиваются переменной `e`.

Листинг 4.2. Префиксные и постфиксные операции инкремента и декремента

```
int i = 5;
int e; // результат выражения
e = i++; // постфиксный инкремент: i имеет значение 6; e имеет значение 5
e = i--; // постфиксный декремент: i имеет значение 5; e имеет значение 6
e = ++i; // префиксный инкремент: i имеет значение 6; e имеет значение 6
e = --i; // префиксный декремент: i имеет значение 5; e имеет значение 5
```

В этом примере операция `i++` возвращает неизмененное значение `5`, которое затем присваивается переменной `e`. Затем значение `i` инкрементируется в качестве побочного эффекта данной операции.

Префиксная операция инкремента увеличивает значение операнда на 1, и лишь потом это увеличенное значение возвращается выражением. Следовательно, выражение `++i` эквивалентно `i = i + 1`, однако `i` вычисляется единожды. В этом примере действие `++i` возвращает инкрементированное значение `6`, которое затем присваивается переменной `e`.

Приоритет и ассоциативность операций

В математике и программировании *порядок выполнения операций* (или приоритет операций) — это набор правил, которые определяют, в каком порядке выполняются операции в заданном выражении. Например, умножение обычно имеет более высокий приоритет по сравнению со сложением. Следовательно, выражение $2 + 3 \times 4$ интерпретируется как $2 + (3 \times 4) = 14$, а не как $(2 + 3) \times 4 = 20$.

Ассоциативность определяет, как группируются операции с одинаковым приоритетом при отсутствии явно указанных скобок. Если соседние операции имеют один и тот же приоритет, то выбор действия, которое нужно выполнить первым, определяется ассоциативностью. *Левоассоциативные*

операции группируют действия слева, а *правоассоциативные* — справа. Группирование можно считать автоматическим добавлением круглых скобок. Например, операция сложения (+) является левоассоциативной, поэтому выражение `a + b + c` интерпретируется как `((a + b) + c)`. Операция присваивания является правоассоциативной, поэтому выражение `a = b = c` эквивалентно `(a = (b = c))`.

В табл. 4.1 перечислены приоритеты и ассоциативность операций в соответствии с синтаксисом языка C[1]. Операции перечислены сверху вниз в порядке уменьшения приоритета.

Таблица 4.1. Приоритеты и ассоциативность операций

Приоритет	Операция	Описание	Ассоциативность
0	(...)	Принудительное группирование	Слева направо
1	++ ––	Постфиксные инкремент и декремент	Слева направо
	()	Вызов функции	
	[]	Обращение к элементу массива	
	.	Обращение к члену структуры или объединения	
	–>	Обращение к члену структуры или объединения через указатель	
	(*тип*) {*список*}	Составной литерал	
2	++ ––	Префиксные инкремент и декремент	Справа налево
	+ –	Унарные плюс и минус	
	! ~	Логическое НЕ и битовое НЕ	
	(*тип*)	Приведение типа	
	*	Разыменовывание	
	&	Взятие адреса	
	sizeof	Размер	
	_Alignof	Выравнивание	
3	* / %	Умножение, деление и остаток	Слева направо
4	+ –	Сложение и вычитание	
5	<< >>	Битовые левый и правый сдвиги	

[1] Основано на таблице «Приоритет операций C», доступной на сайте C++ Reference: https://ru.cppreference.com/w/c/language/operator_precedence.

Приоритет	Операция	Описание	Ассоциативность
6	< <=	Операции сравнения < и ≤	
	> >=	Операции сравнения > и ≥	
7	== !=	Равно и не равно	
8	&	Битовое И	
9	^	Битовое исключающее ИЛИ (XOR)	
10	|	Битовое ИЛИ (включающее ИЛИ)	
11	&&	Логическое И	
12	||	Логическое ИЛИ	
13	?:	Условная операция	Справа налево
14	=	Простое присваивание	
	+= –=	Присваивание через сумму и разность	
	*= /= %=	Присваивание через произведение, частное и остаток	
	<<= >>=	Присваивание через сдвиг влево и сдвиг вправо	
	&= ^= |=	Присваивание через битовые И, исключающее ИЛИ и ИЛИ	
15	,	Объединение выражений в последовательность	Слева направо

Иногда приоритет операций выглядит логично, а иногда не очень. Например, постфиксные операции `++` и `--` имеют более высокий приоритет по сравнению с префиксными `++` и `--`, которые, в свою очередь, имеют тот же приоритет, что и унарная операция `*`. Более того, если `p` — указатель, то `*p++` эквивалентно `*(p++)`, а `++*p` эквивалентно `++(*p)`, поскольку и префиксная операция `++`, и унарная операция `*` являются правоассоциативными. В случае одинаковых приоритета и ассоциативности операции оцениваются слева направо. Правила, по которым определяется порядок выполнения этих операций, показаны в листинге 4.3.

Листинг 4.3. Порядок выполнения операций

```
char abc[] = "abc";
char xyz[] = "xyz";

char *p = abc;
printf("%c", ++*p);

p = xyz;
printf("%c", *p++);
```

В выражении `++*p` сначала происходит разыменование указателя, которое возвращает символ `'a'`. Затем это значение инкрементируется, в результате чего получается символ `'b'`. С другой стороны, в выражении `*p++` указатель сначала инкрементируется, поэтому ссылается на символ `'y'`. Однако результатом *постфиксных* операций инкремента является значение операнда, поэтому исходное значение указателя разыменовывается и мы получаем символ `'x'`. Следовательно, этот код выводит на экран символы `bx`. Чтобы изменить или сделать более понятным порядок выполнения операций, можно воспользоваться скобками `( )`.

Порядок вычисления

Порядок вычисления операндов в любой операции языка C, включая порядок выполнения любых вложенных выражений, обычно не уточняется. Компилятор может вычислить их в любом порядке, который к тому же может меняться при повторном выполнении того же выражения. Такая неопределенность позволяет компилятору генерировать более производительный код за счет выбора наиболее эффективного порядка. Однако этот порядок ограничен приоритетом и ассоциативностью операций.

Листинг 4.4 демонстрирует порядок вычисления аргументов функции. Мы вызываем функцию `max`, которая была определена ранее, передавая ей два аргумента, которые являются результатом вызова функций `f` и `g` соответственно. Порядок выполнения выражений, передаваемых функции `max`, не уточнен, то есть `f` и `g` могут быть вызваны в любом порядке.

Листинг 4.4. Порядок вычисления аргументов функции

```
int glob; // статическое хранилище инициализируется с помощью 0

int f(void) {
  return glob + 10;
}
int g(void) {
  glob = 42;
  return glob;
}
int main(void) {
  int max_value = max(f(), g());
  // ---snip---
}
```

Обе функции (`f` и `g`) обращаются к глобальной переменной `glob`; это значит, они зависят от разделяемого состояния. Когда они вычисляют свои

результаты, значение, передаваемое функции `max`, может изменяться при каждой компиляции. Если функция `f` вызывается первой, то вернет `10`, а если последней, то ее возвращаемым значением будет `52`. Функция `g` всегда возвращает `42`, вне зависимости от порядка вычислений. Таким образом, функция `max` (возвращающая большее из двух значений) может вернуть как `42`, так и `52`, в зависимости от того, в каком порядке были вычислены ее аргументы. Единственная *гарантия* относительно порядка выполнения, которую дает этот код, состоит в том, что `f` и `g` всегда вызываются до `max` и выполняются исключительно по очереди.

Этот код можно переписать так, чтобы он был переносимым и всегда выполнялся предсказуемо:

```
int f_val = f();
int g_val = g();
int max_value = max(f_val, g_val);
```

В этой обновленной программе `f` вызывается для инициализации переменной `f_val`. Это всегда происходит перед выполнением функции `g`, которая вызывается в следующем объявлении для инициализации переменной `g_val`. Если первое вычисление расположено *перед* вторым, то первое из них должно завершиться до того, как начнет выполняться второе. С помощью такой расстановки выражений можно, к примеру, гарантировать, что объект будет записан, прежде чем его начнут считывать в рамках отдельного вычисления. Выполнение `f` всегда происходит до выполнения `g`, поскольку между полным вычислением этих выражений есть точка следования. Точки следования подробно обсуждаются чуть ниже.

Непоследовательные и неопределенно последовательные вычисления

Непоследовательные вычисления могут *перекрывать* друг друга (смешиваться); это значит, их инструкции могут быть выполнены в любом порядке при условии, что последовательность выполнения является *согласованной* — операции чтения и записи производятся в порядке, заданном программой (Лэмпорт, 1979).

Некоторые вычисления имеют *неопределенный порядок*; иными словами, они не могут смешиваться, однако порядок их выполнения может быть каким угодно. Например, следующий оператор выполняет вычисление нескольких значений и имеет побочные эффекты:

```
printf("%d\n", ++i + ++j * --k);
```

Значения `i`, `j` и `k` должны быть прочитаны до того, как их можно будет инкрементировать и декрементировать. Это значит, что чтение `i`, к примеру, должно быть расположено перед побочным эффектом инкремента. Точно так же операция умножения может начаться только после того, как будут получены все побочные эффекты ее операндов. Наконец, ввиду правил, определяющих порядок выполнения, умножение должно закончиться до сложения, равно как и получение всех побочных эффектов для операндов операции `+`. Эти ограничения лишь частично упорядочивают данные операции, поскольку они, к примеру, не требуют, чтобы значение `j` инкрементировалось до декремента `k`. Неупорядоченные вычисления в данном выражении могут выполняться в любом порядке. Это позволяет компилятору переставлять их местами и кэшировать значения в регистрах для повышения общей производительности. А вот функции упорядочены неопределенно, и их выполнение не перемежается.

Точки следования

Точка следования — момент завершения всех побочных эффектов. Эти точки косвенно определяются языком, но вы можете изменять их местоположение, по-разному описывая логику своей программы.

Точки следования перечислены в приложении C к стандарту C. Они возникают между вычислением двух *полных выражений* (таких, которые не являются частью другого выражения или объявления). Точка следования также находится на входе в вызванную функцию и на выходе из нее.

Если один побочный эффект не упорядочен относительного другого, относящегося к тому же скаляру или вычислению, которое использует значение того же скалярного объекта, то поведение кода не определено. *Скалярным типом* может быть либо арифметический тип, либо тип указателя. Выражение `i++ * i++` в следующем фрагменте кода проводит две неупорядоченные операции с `i`:

```
int i = 5;
printf("Result = %d\n", i++ * i++);
```

Вы можете подумать, что этот код выведет значение `30`, однако из-за его неопределенного поведения подобный результат не гарантирован. По меньшей мере у нас есть возможность сделать так, чтобы побочные

эффекты завершались до того, как будет прочитано значение; для этого каждую операцию с побочным эффектом нужно оформить в виде полного выражения. Чтобы избавиться от неопределенного поведения, мы можем переписать этот код следующим образом:

```
int i = 5;
int j = i++;
int k = i++;
printf("Result = %d\n", j * k);
```

Теперь данный код содержит точку следования между каждой операцией с побочным эффектом. Однако невозможно с уверенностью сказать, представляет ли эта переписанная версия изначальный замысел программиста, поскольку у оригинального кода нет четко определенного значения. Если вы решите отказаться от точек следования, то убедитесь в том, что полностью понимаете порядок, в котором происходят побочные эффекты. Этот же код можно записать следующим образом, не меняя его поведение:

```
int i = 5;
int j = i++;
printf("Result = %d\n", j * i++);
```

Итак, мы рассмотрели механизм выполнения выражений. Теперь вернемся к обсуждению отдельных операций, начиная с `sizeof`.

Операция sizeof

Вы можете использовать операцию `sizeof` для определения размера ее операнда (в байтах); в частности, она возвращает беззнаковое целое значение типа `size_t`, которое представляет размер. Эта информация необходима для выполнения большинства действий с памятью, включая выделение и копирование. Тип `size_t` определен в `<stddef.h>` и других заголовочных файлах. Чтобы скомпилировать любой код, в котором фигурирует `size_t`, необходимо подключить один из этих файлов.

Операции `sizeof` можно передать невычисленное выражение полного объектного типа или его имя, заключенное в круглые скобки:

```
int i;
size_t i_size = sizeof i;      // размер объекта i
size_t int_size = sizeof(int); // размер типа int
```

Размещение операндов `sizeof` в скобках всегда безопасно, поскольку не влияет на способ вычисления их размеров. Результат вызова операции `sizeof` представляет собой константное выражение, если только операнд не является массивом переменной длины. Операнд `sizeof` не вычисляется[1].

Если вам нужно определить количество доступных для хранения битов, то можете умножить размер объекта на `CHAR_BIT`, что даст вам количество битов, содержащихся в байте. Например, выражение `CHAR_BIT * sizeof(int)` вернет количество битов в объекте типа `int`.

Все объектные типы, кроме символьных, могут содержать не только биты для представления значения, но и дополнительные биты заполнения. Разные целевые платформы могут по-разному упаковывать байты в многобайтные машинные слова. Отличия состоят в *порядке байтов*[2] и приводят к тому, что для передачи объектов по сети вам следует согласовать общий внешний формат представления объектов и использовать функции для преобразования объектов в платформозависимом представлении в этот общий формат и обратно (осуществлять маршалинг и демаршалинг).

Арифметические операции

Далее описываются несколько операций, производящих арифметические действия с числовыми типами. Некоторые из них можно также применять и к неарифметическим операндам.

Унарные операции + и –

Унарные операции + и – работают с одиночными операндами числовых типов. Операция - возвращает отрицательное значение своего операнда (как будто операнд умножили на –1). Унарная операция + просто возвращает значение. Обе они существуют в основном для выражения положительных и отрицательных чисел.

---

[1] Но не всегда. Если для вычисления размера выражение нужно выполнить, то оно будет выполнено; например, sizeof(int[i++]) будет инкрементить i, потому что здесь вычисляется размер массива переменной длины. — *Примеч. науч. ред.*

[2] В английском языке используется термин endianness, позаимствованный из сатирического романа Джонатана Свифта «Путешествия Гулливера», впервые опубликованного в 1726 году. В романе описывается война, разразившаяся из-за спора о том, с какого конца надо разбивать вареные яйца — тупого или острого.

Если операнд имеет малый целочисленный тип, то его разрядность повышается (см. главу 3) и результат операции получает расширенный тип. Исторически так сложилось, что в языке C нет отрицательных целочисленных констант; такое значение, как –25, на самом деле является *r*-значением типа `int` величиной 25, перед которым указана унарная операция `-`.

Логическая операция отрицания

Унарная *логическая операция отрицания* (`!`) возвращает:

- 0, если значение операнда не равно 0;
- 1, если значение операнда равно 0.

Операнд имеет скалярный тип, а результат — тип `int` (по историческим причинам). Выражение `!E` эквивалентно `(0 == E)`. Операцию логического отрицания часто используют для проверки на нулевые указатели; например, `!p` является эквивалентом `(NULL == p)`.

Мультипликативные операции

К двоичным *мультипликативным операциям* относят умножение (`*`), деление (`/`) и взятие остатка (`%`). Для подбора общего типа к мультипликативным операндам автоматически применяются обычные арифметические преобразования. Вы можете умножать и делить как целочисленные, так и значения с плавающей запятой, однако операция взятия остатка работает только с целочисленными операндами.

Разные языки программирования поддерживают разные виды операций целочисленного деления, включая деление с остатком, деление с возвращением наименьшего целого и деление с усечением. При *евклидовом делении* остаток всегда получается неотрицательным (Боут, 1992). При *делении с возвращением наименьшего целого* частное округляется в сторону минус бесконечности (Кнут, 1997). При *делении с усечением* результатом операции `/` является алгебраическое частное без какой-либо дробной части. Это действие часто называют *усечением в сторону нуля*.

В языке программирования C реализовано деление с усечением. Это значит, что остаток всегда имеет тот же знак, что и делимое, как показано в табл. 4.2.

Таблица 4.2. Деление с усечением

/	**Частное**	%	**Остаток**
10 / 3	3	10 % 3	1
10 / –3	–3	10 % –3	1
–10 / 3	–3	–10 % 3	–1
–10 / –3	3	–10 % –3	–1

В целом если частное `a / b` представимо, то выражение `(a / b) * b + + a % b` эквивалентно `a`. В противном случае если значение делителя равно 0 или `a / b` приводит к переполнению, то операции `a / b` и `a % b` приводят к неопределенному поведению.

Чтобы избежать сюрпризов, стоит должным образом разобраться в поведении операции `%`. Например, в следующем коде определена дефектная функция `is_odd`, которая пытается проверить, является ли целое значение нечетным:

```
bool is_odd(int n) {
  return n % 2 == 1;
}
```

Поскольку результат операции `%` всегда имеет тот же знак, что и делимое `n`, когда `n` отрицательное и нечетное, `n % 2` возвращает `-1` и результат функции равен `false`. Альтернативное и корректное решение состоит в проверке того, не равен ли остаток 0 (так как остаток нуля всегда один и тот же, независимо от делимого):

```
bool is_odd(int n) {
  return n % 2 != 0;
}
```

Во многих процессорах взятие остатка реализовано как часть операции деления, что может привести к переполнению, если делимое равно минимальному отрицательному значению целочисленного типа со знаком, а делитель равен –1. Это происходит вопреки тому, что в математическом смысле такая операция `%` должна вернуть 0.

Стандартная библиотека C предоставляет функции для взятия остатка, усечения и округления значений с плавающей запятой, в том числе функцию `fmod`.

Аддитивные операции

К бинарным *аддитивным операциям* относят сложение (+) и вычитание (-). Эти команды можно применять к двум операндам числовых типов, но с их помощью также можно выполнять арифметические действия с указателями. Последние будут рассмотрены ближе к концу текущей главы; данное же обсуждение ограничивается операциями с числовыми типами.

Бинарная операция + суммирует два своих операнда. Бинарная операция – вычитает правый операнд из левого. К операндам арифметических типов в обеих операциях применяются обычные арифметические преобразования.

Битовые операции

Битовые операции используются для управления битами объекта или любого целочисленного выражения. Их, как правило, применяют к объектам, представляющим *битовые карты*: каждый бит сигнализирует о том, что нечто «включено» или «выключено», «установлено» или «сброшено» (или какое-то другое двоичное отношение).

Битовые операции (| & ^ ~) работают с битами как с сугубо двоичной моделью, не придавая значения тому, что именно представлено этими битами:

```
  1 1 0 1 = 13
^ 0 1 1 0 = 6
= 1 0 1 1 = 11
```

Битовые карты лучше всего описываются с помощью беззнаковых целочисленных типов, поскольку разряд, отведенный знаку, можно более эффективно использовать в виде значения внутри карты. К тому же операции с такими типами менее склонны к неопределенному поведению.

Операция дополнения

Унарная операция дополнения (битовой инверсии) (~) принимает один операнд целочисленного типа и возвращает его *битовое дополнение*, то есть инвертирует каждый бит исходного значения. Операция дополнения используется, к примеру, для применения `umask` в POSIX. Права доступа к файлу являются результатом логической операции И по отношению к дополнению маски и запрошенных прав доступа. Операнд проходит

через повышение разрядности, и результат имеет расширенный тип. Так, следующий фрагмент кода применяет операцию ~ к значению типа `unsigned char`:

```
unsigned char uc = UCHAR_MAX; // 0xFF
int i = ~uc;
```

В архитектуре с восьмибитным типом `char` и 32-битным типом `int` переменной `uc` присваивается значение `0xFF`. При использовании в качестве операнда для операции ~ переменная `uc` расширяется до 32-битного типа `signed int` за счет добавления нулей. Унарное дополнение выглядит как `0xFFFFFF00`. Следовательно, на данной платформе дополнение типа `unsigned short` всегда имеет вид отрицательного значения типа `signed int`. Во избежание подобных сюрпризов следует взять за правило, что битовые операции должны выполняться с одним беззнаковым целочисленным типом достаточной ширины.

Операции сдвига

Операции сдвига перемещают значение каждого бита операнда целочисленного типа на указанное количество позиций. Эти операции часто используются в системном программировании, где повсеместно применяются битовые маски. Операции также могут пригодиться для упаковки и распаковки данных в коде, который работает с сетевыми протоколами или файловыми форматами. Поддерживаются операции сдвига влево вида:

сдвигаемое_выражение `<<` *аддитивное_выражение*

и сдвига вправо вида:

сдвигаемое_выражение `>>` *аддитивное_выражение*

Сдвигаемое_выражение — значение, которое нужно сдвинуть, а *аддитивное выражение* — количество битов, на которое сдвигается значение. На рис. 4.1 показан логический сдвиг влево на один бит.

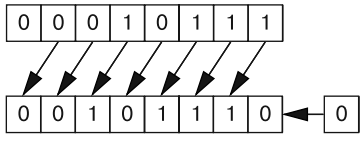

Рис. 4.1. Логический сдвиг влево на один бит

Аддитивное выражение определяет количество битов, на которое нужно сдвинуть значение. Например, результатом `E1 << E2` будет значение `E1`, сдвинутое влево на `E2` позиций; освобожденные биты заполняются нулями. Если `E1` имеет беззнаковый тип, то результат будет равен $E1 \times 2^{E2}$. Если значение не может быть представлено с помощью итогового типа, то приводится по модулю к величине, на единицу большей, чем максимально допустимое значение. Если `E1` имеет знаковый тип и неотрицательное значение, а выражение $E1 \times 2^{E2}$ можно представить с помощью итогового типа, то оно и станет результатом; в противном случае поведение не определено. Точно так же результатом `E1 >> E2` является значение `E1`, сдвинутое вправо на `E2` позиций. Если `E1` имеет беззнаковый тип или тип со знаком, но с неотрицательным значением, то результатом будет целая часть частного $E1 / 2^{E2}$. Если `E1` имеет знаковый тип и отрицательное значение, то результат определяется реализацией и может иметь вид как арифметического (расширенного с учетом знака), так и логического (беззнакового) сдвига. Это показано на рис. 4.2.

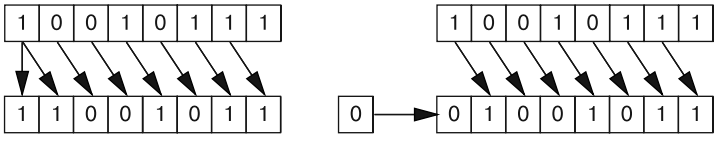

Рис. 4.2. Арифметический сдвиг вправо (со знаком) и логический (беззнаковый) сдвиг вправо на один бит

В обоих сдвигах целочисленные операнды проходят через повышение разрядности; каждый из них имеет целочисленный тип. Результат имеет тот же тип, что и расширенный левый операнд. Обычные арифметические преобразования *не* проводятся.

Сдвиги влево и вправо можно использовать для умножения и деления на степень двойки, но это не очень удачный подход. Лучше всего применять соответствующие операции и позволить компилятору при необходимости оптимизировать вычисления за счет подстановки вместо них операций сдвига. Самостоятельное выполнение такой подстановки является примером преждевременной оптимизации. Дональд Кнут, автор книги *«Искусство программирования»* (1997), назвал преждевременную оптимизацию «корнем всех проблем».

Количество сдвигаемых битов не должно быть отрицательным, больше ширины расширенного левого операнда или равно ей, поскольку это

приводит к неопределенному поведению. В листинге 4.5 показано, как выполнять сдвиг вправо для целочисленных значений со знаком и без, не допуская подобных ошибок.

Листинг 4.5. Правильные операции сдвига вправо

```
extern int si1, si2, sresult;
extern unsigned int ui1, ui2, uresult;
// ---snip---
❶   if ( (si2 < 0) || (si2 >= sizeof(int)*CHAR_BIT) ) {
  /* ошибка */
}
else {
  sresult = si1 >> si2;
}
❷ if (ui2 >= sizeof(unsigned int)*CHAR_BIT) {
  /* ошибка */
}
else {
  uresult = ui1 >> ui2;
}
```

Если целое значение имеет знак ❶, то вы должны убедиться в том, что количество сдвигаемых битов не является отрицательным, больше ширины расширенного левого операнда или равным ей. В случае с беззнаковыми целочисленными типами ❷ проверку на отрицательные значения можно пропустить, поскольку такие типы не могут быть отрицательными. Похожим образом можно проводить и операции сдвига влево.

Битовое И

Двоичная операция битового И (`&`) возвращает битовое И для двух операндов целочисленного типа. К обоим операндам применяются обычные арифметические преобразования. Каждый бит результата устанавливается тогда и только тогда, когда задан каждый из соответствующих битов в преобразованных операндах. Это показано в табл. 4.3.

Таблица 4.3. Таблица истинности битового И

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

Битовое исключающее ИЛИ

Бинарная операция битового исключающего ИЛИ (^) возвращает битовое исключающее ИЛИ для двух целочисленных операндов. Иначе говоря, каждый бит результата устанавливается тогда и только тогда, когда задан лишь один из соответствующих битов в преобразованных операндах (табл. 4.4). Это можно перефразировать как «либо тот, либо другой, но не оба сразу»).

Таблица 4.4. Таблица истинности битового исключающего ИЛИ

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

Принимая во внимание циклический перенос, исключающее ИЛИ эквивалентно операции сложения целых значений по модулю 2, то есть 1 + 1 mod 2 = 0 (Левин, 2012). Операнды должны быть целочисленными, и к ним обоим применяются обычные арифметические преобразования.

Новички часто путают операцию исключающего ИЛИ с операцией возведения в степень, ошибочно полагая, что выражение `2 ^ 7` возводит 2 в степень 7. Вместо этого в языке C предусмотрены функции `pow`[1], определенные в заголовочном файле `<math.h>`. Вы можете видеть их в листинге 4.6. Функции `pow` принимают аргументы с плавающей запятой и возвращают результат того же типа, поэтому имейте в виду, что из-за усечения и других погрешностей они могут вернуть не то, что вы ожидаете.

Листинг 4.6. Использование функций pow

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

int main(void) {
  int i = 128;
  if (i == pow(2, 7)) {
    puts("equal");
  }
}
```

[1] Имеются в виду функции pow (для double), powf(для float) и powl(для long double). — *Примеч. науч. ред.*

Этот код вызывает функцию `pow`, чтобы возвести 2 в степень 7. Поскольку 2^7 равно 128, а также учитывая, что число 128 может быть точно представлено типом `double`, данная программа выведет `equal`.

Битовое включающее ИЛИ

Двоичная операция битового включающего ИЛИ (`|`) возвращает битовое включающее ИЛИ для целочисленных операндов. Операнды должны быть целочисленными, и к ним обоим применяются обычные арифметические преобразования. Каждый бит результата устанавливается тогда и только тогда, когда задан хотя бы один из соответствующих битов в преобразованных операндах. Это показано в табл. 4.5.

Таблица 4.5. Таблица истинности включающего ИЛИ

x	y	x | y
0	0	0
0	1	1
1	0	1
1	1	1

Логические операции

Операции логического И (`&&`) и *ИЛИ* (`||`) используются в основном для логического объединения двух и более выражений скалярных типов. Их часто применяют для проверки условий, таких как первый операнд условной операции, управляющее выражение оператора `if` и управляющее выражение цикла `for`, и для объединения нескольких сравнений. Логические операции не следует использовать в сочетании с битовыми картами, поскольку они в первую очередь рассчитаны на булеву логику.

Операция `&&` возвращает 1, если ни один из операндов не равен 0; в противном случае возвращается 0. С точки зрения логики это означает, что выражение `a && b` является истинным, если `a` и `b` истинны.

Операция `||` возвращает 1, если хотя бы один из операндов не равен 0; в противном случае возвращается 0. С точки зрения логики это означает,

что выражение `a || b` является истинным, если истинен один из операндов, `a` или `b`, или оба сразу.

В стандарте C обе команды определены с точки зрения неравенства нулю, поскольку значения операндов могут быть равны не только 0 или 1. Обе операции принимают операнды скалярных типов (целочисленных и с плавающей запятой) и указатели, а результат операции имеет тип `int`.

В отличие от соответствующих двоичных битовых операций, логические И и ИЛИ гарантируют порядок вычисления слева направо; между вычислением второго и первого операндов находится точка следования.

Логические операции проводят вычисления *по короткой схеме*: если результат можно получить за счет вычисления только первого операнда, то второй операнд не вычисляется. Например, выражение `0 && unevaluated` возвращает 0 независимо от значения `unevaluated`, поскольку последнее, каким бы оно ни было, не может повлиять на результат. Учитывая это, значение операнда `unevaluated` не вычисляется. То же самое относится и к `1 || unevaluated`, поскольку это выражение всегда возвращает 1.

Вычисления по короткой схеме часто используются в операциях с указателями:

```
bool isN(int* ptr, int n){
  return ptr && *ptr == n; // не разыменовывайте нулевой указатель
}
```

Данный код проверяет значение `ptr`. Если `ptr` равно `NULL`, то второй операнд в `&&` не вычисляется, что предотвращает разыменовывание нулевого указателя.

Это позволяет избежать ненужных вычислений. Ниже предикативная функция `is_file_ready` возвращает `true`, если файл готов:

```
is_file_ready() || prepare_file()
```

В этом случае если функция `is_file_ready` возвращает `true`, то второй операнд в `||` не вычисляется, поскольку файл уже не нужно подготавливать. Таким образом, мы избегаем лишних вычислений при условии, что проверка готовности файла менее затратная, чем его подготовка (и вероятность готовности файла довольно велика).

В случае, когда второй операнд имеет побочные эффекты, необходимо проявлять бдительность, поскольку может быть не совсем очевидно, возникают ли они. Например, в следующем фрагменте кода значение `i` инкрементируется, только если `i >= 0`:

```
enum { max = 15 };
int i = 17;

if ( (i >= 0) && ( (i++) <= max) ) {
  // ---snip---
}
```

Этот код может быть корректным, но, скорее всего, в нем допущена ошибка.

Операции приведения типов

Операции приведения типов позволяют явно преобразовать значение из одного типа в другой. Чтобы выполнить приведение, нужно перед выражением написать имя типа в круглых скобках; в результате значение выражения приводится к неквалифицированной версии заданного типа. Следующий код иллюстрирует явное приведение `x` из типа `double` к типу `int`:

```
double x = 1.2;
int sum = (int)x + 1; // Явное приведение double к int
```

Указываемый тип должен быть квалифицированным или неквалифицированным скалярным типом (исключение составляет тип `void`). Операнд тоже должен иметь скалярный тип; указатель не может быть приведен ни к какому типу с плавающей запятой, и наоборот.

Приведение типов — чрезвычайно мощная операция, которую нужно использовать с осторожностью. Например, она может по-новому интерпретировать имеющиеся биты в качестве значения заданного типа, не меняя их содержимое:

```
intptr_t i = (intptr_t)some_pointer; // заново интерпретируем биты
                                     // как целое число
```

Эти биты также могут принять тот вид, который необходим для представления исходного значения с помощью итогового типа:

```
int i = (int)some_float; // приводим биты к целочисленному представлению
```

Приведение типов может намеренно предотвращать вывод диагностических сообщений. Взгляните, к примеру, на следующий фрагмент кода:

```
char c;
// ---snip---
while ((c = fgetc(in)) != EOF) {
  // ---snip---
}
```

Visual C++ 2019 с уровнем предупреждений `/W4` сгенерирует при его компиляции следующее сообщение:

```
Severity Code Description
Warning C4244 '=': conversion from 'int' to 'char', possible loss of data
```

Если же привести значение к типу `char`, то от предупреждения можно избавиться, но проблема при этом останется:

```
char c;
while ((c = (char)fgetc(in)) != EOF) {
  // ---snip---
}
```

В целях минимизации этих рисков в C++ предусмотрены собственные, менее мощные механизмы приведения типов.

Условная операция

В языке C только *условная операция* (`?:`) принимает сразу три операнда[1]. Ее результат зависит от условия. Условная операция используется следующим образом:

```
результат = условие ? результат_если_истинное : результат_если_ложное
```

Условная операция вычисляет первый операнд, который называют *условием*. Затем, если условие истинное, то вычисляется второй операнд (*результат_если_истинное*), а если ложное — то третий (*результат_если_ложное*). Результатом является значение либо второго, либо третьего операнда (в зависимости от того, какой из них был вычислен).

Данный результат приводится к общему типу, основанному на типах второго и третьего операндов. Между вычислением первого и второго

[1] Поэтому в русской терминологии его часто называют просто тернарным оператором. — *Примеч. науч. ред.*

или третьего операндов (в зависимости от того, какой из них вычисляется) находится точка следования, поэтому компилятор проследит за тем, чтобы все побочные эффекты завершились до того, как будет вычислен второй или третий операнд.

Условная операция похожа на блок управления потоком выполнения `if-else`, но при этом возвращает значение, словно функция. В отличие от блока `if-else` условная операция позволяет инициализировать объекты с квалификатором `const`:

```
const int x = (a < b) ? b : a;
```

Первый операнд условной операции должен иметь скалярный тип. Типы второго и третьего операндов должны быть совместимы (грубо говоря). Более подробную информацию об ограничениях этой операции и о том, как именно определяется возвращаемый тип, можно найти в разделе 6.5.15 стандарта C (ISO/IEC 9899:2018).

Операция _Alignof

Операция `_Alignof` возвращает целочисленную константу, представляющую требования к выравниванию объявленного полного объектного типа его операнда. Операнд не вычисляется. Если указанный тип — массив, то данная операция возвращает требования к выравниванию типа элементов. `_Alignof` используется в виде удобного макроса `alignof`, доступного в заголовочном файле `<stdalign.h>`. Она подходит как для статических, так и для динамических утверждений, с помощью которых проверяются предположения о программе (более подробно об этом поговорим в главе 11). Эти утверждения позволяют диагностировать ситуации, в которых ваши предположения ошибочны. В листинге 4.7 показано использование операции `_Alignof` и макроса `alignof`.

Листинг 4.7. Использование операции _Alignof

```
#include <stdio.h>
#include <stddef.h>
#include <stdalign.h>
#include <assert.h>

int main(void) {
  int arr[4];
```

```
  static_assert(_Alignof(arr) == 4, "unexpected alignment");
  // статическое утверждение
  assert(alignof(max_align_t) == 16); // динамическое утверждение
  printf("Alignment of arr = %zu\n", _Alignof(arr));
  printf("Alignment of max_align_t = %zu\n", alignof(max_align_t));
}
```

Эта простая программа не делает ничего особо полезного. Она объявляет массив `arr` из четырех элементов `int` и затем делает статическое утверждение о выравнивании массива и динамическое утверждение о выравнивании `max_align_t` (объектного типа с максимально необходимым выравниванием). Дальше эти значения выводятся на экран. Если динамические утверждения включены, то программа может не скомпилироваться из-за `static_assert`, не выполнить динамическое утверждение, ничего не вывести или вывести следующее:

```
Alignment of arr = 4
Alignment of max_align_t = 16
```

Операции сравнения

К *операциям сравнения* относятся `==` (равно), `!=` (не равно), `<` (меньше), `>` (больше), `<=` (меньше или равно) и `>=` (больше или равно). В случае истинности заданного сравнения каждая из них возвращает 1, а в случае ложности — 0. Опять же по историческим причинам результат имеет тип `int`.

Стоит отметить, что язык C не интерпретирует выражение `a < b < c` как «`b` больше `a`, но меньше `c`», как принято в математике. Вместо этого оно эквивалентно `(a < b) < c`, что означает следующее: если `a` меньше `b`, то компилятор должен сравнить `1` и `c`, а если нет — то `0` и `c`. Если это то, что вам нужно, то желательно добавить круглые скобки, чтобы прояснить ситуацию для тех, кто впоследствии может просматривать ваш код. Некоторые компиляторы, включая GCC и Clang, предоставляют флаг `-Wparentheses` для диагностики таких проблем. Если же вам нужно определить, является ли `b` больше `a`, но меньше `c`, то это можно записать как `(a < b) && (b < c)`.

Операции равенства и неравенства имеют меньший приоритет, чем другие операции сравнения, — предположения об обратном являются распространенной ошибкой. Это значит, что выражение `a < b == c < d` вычисляется так же, как и `(a < b) == (c < d)`. В обоих случаях сравнения `a < b` и `c < d` вычисляются первыми, а затем их результаты (0 или 1) проверяются на равенство.

С помощью этих операций можно сравнивать арифметические типы и указатели. При сравнении двух указателей результат зависит от того, как объекты, на которые они указывают, размещены в адресном пространстве друг относительно друга. Если оба они указывают на один и тот же объект, то их считают равными.

Операции равенства и неравенства отличаются от других операций сравнения. Например, последние нельзя применять к двум указателям на разные объекты, поскольку делать это бессмысленно:

```
int i, j;
bool b1 = &i < &j;  // неопределенное поведение
bool b2 = &i == &j; // сойдет, хоть и тавтология
```

Операции составного присваивания

Операции составного присваивания изменяют текущее значение объекта, выполняя с ним какое-то действие. Они перечислены в табл. 4.6.

Таблица 4.6. Операции составного присваивания

Операция	Описание
+= –=	Присваивание через сумму и разность
*= /= %=	Присваивание через произведение, частное и взятие остатка
<<= >>=	Присваивание через сдвиг влево и сдвиг вправо
&= ^= |=	Присваивание через битовые И, исключающее ИЛИ и ИЛИ

Составное присваивание вида `E1` *op* `= E2` эквивалентно выражению простого присваивания `E1 = E1` *op* `(E2)`, если не считать того, что `E1` вычисляется только один раз. Составное присваивание в основном используется в качестве краткой записи и не поддерживает логические операции.

Операция «запятая»

В языке C запятые используются двумя способами: в качестве операций и средства разделения элементов в списках (таких как аргументы функции или цепочки объявлений). *Операция «запятая»* (`,`) позволяет вычислить одно выражение перед другим. Сначала операнд, находящийся по левую

сторону от запятой, вычисляется как выражение `void` (то есть его результат отбрасывается. — *Примеч. науч. ред.*). Между вычислением левого и правого операндов находится точка следования. Далее вычисляется правый операнд. Результат операции «запятая» имеет тип и значение правого операнда — в основном ввиду того, что это последнее вычисленное выражение.

Эту операцию нельзя использовать в контекстах, в которых запятая может разделять элементы списка. Чтобы обойти это ограничение, выражения, разделяемые запятыми, можно заключить в круглые скобки. Вы также можете поместить запятую во второе выражение условной операции. Например, следующий вызов функции принимает три параметра:

```
f(a,❶ (t=3,❷ t+2),❸ c)
```

Первая запятая ❶ разделяет первый и второй аргументы функции. Вторая запятая ❷ является операцией. Сначала выполняется присваивание, а затем сложение. Благодаря точке следования сложение гарантированно начинается после завершения присваивания. Результат операции имеет те же тип (`int`) и значение (`5`), что и правый операнд. Третья запятая ❸ разделяет второй и третий аргументы функции.

Арифметические операции с указателями

Ранее в этой главе я упомянул о том, что аддитивные операции (сложение и вычитание) можно использовать для работы как с арифметическими значениями, так и с указателями на объекты. В данном разделе мы рассмотрим сложение указателя и целого числа, вычисление разности двух указателей и вычитание из указателя целочисленного значения.

Сумма или разность выражения целочисленного типа и указателя возвращает значение того же типа, что и указатель. Если указатель ссылается на элемент массива, то результат указывает на смещение относительно данного элемента. Если итоговый указатель выходит за пределы массива, то это приводит к неопределенному поведению. Разница между индексами исходного и итогового элементов равна целочисленному выражению:

```
int arr[100];
int *arrp1 = arr[40];
int *arrp2 = arrp1 + 20;      // arrp2 указывает на arr[60]
printf("%td\n", arrp2-arrp1); // выводит 20
```

В языке C можно сформировать указатель на любой элемент массива, в том числе и на такой, который указывает на несуществующий элемент, следующий за последним элементом (также известный как «слишком дальний указатель»). Это может показаться необычным и лишним, но во многих старых программах инкрементируемый указатель становится *слишком дальним*, и комитет во главе стандарта C не хотел объявлять весь этот код некорректным (к тому же такое поведение вполне типично в итераторах C++). На рис. 4.3 показано формирование указателя, который становится слишком дальним.

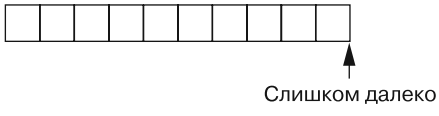

Рис. 4.3. Указатель заходит на одну позицию дальше последнего элемента в объекте массива

Если операнд-указатель и результат операции указывают на элементы одного и того же массива или оба являются слишком дальними указателями, то вычисление не переполняется; в противном случае поведение не определено. Чтобы соблюсти требование касательно слишком дальних указателей, реализации достаточно предоставить один дополнительный байт (который может принадлежать другому объекту из этой же программы), идущий сразу после объекта.

Язык C также позволяет обращаться с объектами как с массивами, которые содержат всего один элемент. Благодаря этому вы можете получить слишком дальний указатель из скалярного значения.

Это дает возможность двигать указатель до тех пор, пока он не станет равным указателю на байт, идущий за объектом. Например:

```
int m[2] = {1, 2};

int sum_m_elems(void) {
  int *pi; int j = 0;
  for (pi = &m[0]; pi < &m[2]; ++pi) j += *pi;
  return j;
}
```

Здесь оператор `for` (более подробно о нем поговорим в следующей главе) в функции `sum_m_elems` выполняет цикл, пока `pi` меньше адреса слишком

дальнего указателя, выходящего за пределы массива `m`. Указатель `pi` инкрементируется в конце каждой итерации цикла, пока не станет слишком дальним, в результате чего условие цикла станет равным 0.

При вычитании одного указателя из другого они оба должны ссылаться на элементы одного и того же массива или на слишком дальний элемент. Эта операция возвращает разность индексов двух элементов. Результат имеет тип `ptrdiff_t` (знаковый целочисленный тип). При выполнении вычитания следует быть внимательными, поскольку диапазона `ptrdiff_t` может не хватить для представления разности указателей на элементы очень больших массивов `char`.[1] Арифметические операции с указателями не работают с отдельными байтами, а автоматически *подстраиваются* под размер элементов.

Резюме

В этой главе вы научились использовать операции для написания простых выражений, которые работают с различными типами объектов. Заодно познакомились с рядом фундаментальных понятий языка C, такими как *l*- и *r*-значения, вычисления значений и побочные эффекты, определяющими, как вычисляются выражения. Кроме того, вы узнали, каким образом приоритеты операций, ассоциативность, порядок вычисления, точки следования и чередование в случае операции объединения могут повлиять на общий порядок выполнения программы.

В следующей главе речь пойдет об управлении выполнением кода с помощью операторов выбора, итерирования и перехода.

[1] Больше, чем PTRDIFF_MAX. И если ptrdiff_t не хватает для представления разности, то поведение не определено (перечислено в приложении J.2 к стандарту). — *Примеч. науч. ред.*

5 Управляющая логика

В этой главе вы узнаете, как управлять порядком вычисления отдельных выражений. Вначале мы пройдемся по операторам-выражениям и составным операторам, которые определяют, какая работа будет выполняться. Затем рассмотрим три вида операторов, определяющих, какие блоки кода выполняются и в каком порядке это происходит: операторы выбора, итерирования и перехода.

Операторы-выражения

Оператор-выражение — произвольное выражение, которое заканчивается точкой с запятой. Это один из самых распространенных видов выражений, представляющий элементарную операцию. Примеры операторов-выражений показаны в листинге 5.1.

Листинг 5.1. Типичные операторы-выражения

```
a = 6;
c = a + b;
; // нулевое выражение, ничего не делает
++count;
```

Первый оператор состоит из выражения, которое присваивает значение переменной `a`. Выражение во втором операторе присваивает сумму `a` и `b` переменной `c`. Третий оператор является нулевым; его можно задействовать в ситуациях, когда синтаксис языка требует наличия оператора, но вам не нужно вычислять никакое выражение. Нулевые операторы часто

используются в качестве заполнителей при итерировании или для назначения меток в конце составных операторов или функций. Четвертый оператор — это выражение, которое инкрементирует значение `count`.

После вычисления каждого полного выражения его значение (если таковое имеется) отклоняется (это относится и к выражениям присваивания, в которых само присваивание является побочным эффектом операции), чтобы любой полезный результат был следствием побочных эффектов (как обсуждалось в главе 4). Три из четырех операторов-выражений в показанном выше примере имеют побочные эффекты (нулевой оператор ничего не делает). По завершении всех эффектов выполнение программы переходит к оператору, который идет за точкой с запятой.

Составные операторы

Составной оператор (или *блок*) — это список из любого количества операторов (начиная с нуля), заключенных в фигурные скобки. В блоке могут находиться любого рода операторы, которые были описаны в данной главе. Некоторые из них могут быть объявлениями (в ранних версиях C объявления внутри блока должны были находиться перед любыми другими операторами, но это ограничение больше не актуально). Все операторы в блоке выполняются последовательно, если только на их работу не влияет управляющий оператор. После того как будет вычислен последний из них, выполнение переходит к коду, который находится сразу за закрывающей фигурной скобкой:

```
{
  static int count = 0;
  c += a;
  ++count;
}
```

В этом примере объявляется статическая переменная `count` типа `int`. Во второй строчке переменная `c`, объявленная во внешней области видимости, увеличивается на значение, хранящееся в `a`. В конце инкрементируется `count`, чтобы подсчитать, сколько раз выполнялся этот блок.

Составные операторы могут быть вложенными — это когда один составной оператор полностью охватывает другую. У вас также могут быть блоки без каких-либо операторов (просто пустые фигурные скобки).

Стиль кода

В различных стилях кода фигурные скобки размещаются по-разному. Если вы редактируете уже существующий проект, то будет разумно следовать используемому в нем стилю. В противном случае посмотрите, в каких стилях пишут свой код опытные программисты на C, и выберите тот, который кажется вам наиболее ясным. Например, некоторые программисты выравнивают открывающие и закрывающие скобки по горизонтали, чтобы каждой из них было легче найти пару. Другие придерживаются стиля, применяемого в книге Брайана Кернигана и Денниса Ритчи «Язык программирования Си» (1988), где открывающая скобка находится в конце предыдущей строки, а закрывающей выделяется отдельная строчка. Выберите какой-то один стиль и всегда ему следуйте.

Операторы выбора

Операторы выбора позволяют выполнять вложенные операторы в зависимости от значения *управляющего выражения*. Оно определяет, какие операторы выполняются, руководствуясь условием. Это позволяет писать код, который генерирует разный вывод при получении разного ввода. К операторам выбора относятся `if` и `switch`.

Оператор if

Оператор `if` позволяет программистам выполнять другие операторы в зависимости от значения управляющего выражения скалярного типа.

Существует два вида операторов `if`. Первый определяет, выполняется ли вложенный оператор:

```
if (выражение)
  подоператор
```

В этом случае *подоператор* выполняется, если *выражение* не равно 0. Условному выполнению подлежит только один вложенный в `if` оператор, хотя он может быть составным.

В листинге 5.2 показана функция деления, которая использует операторы `if`. Она принимает заданные делимое и делитель и возвращает

результат их деления в объект, на который ссылается `quotient`. Функция выполняет проверку как на нуль, так и на переполнение целочисленного значения со знаком; если хотя бы одна из них не пройдена, то возвращается `false`.

Листинг 5.2. Функция безопасного деления

```
bool safediv(int dividend, int divisor, int *quotient) {
❶ if (!quotient) return false;
❷ if ((divisor == 0) || ((dividend == INT_MIN) && (divisor == -1)))
  ❸ return false;
❹ *quotient = dividend / divisor;
return true;
}
```

Первая строчка этой функции ❶ проверяет, не равна ли переменная `quotient` нулю. Если `quotient` имеет нулевое значение, то функция возвращает `false`, сигнализируя о том, что ей не удается вернуть значение. Оператор `return` рассматривается позже в данной главе.

Вторая строчка функции ❷ содержит более сложный оператор `if`. Ее управляющее выражение проверяет, равен ли делитель нулю и приведет ли деление к переполнению целочисленного типа со знаком. Если результат этого выражения не равен нулю, то функция возвращает `false` ❸, сигнализируя о невозможности получить результат деления. Если управляющее выражение оператора `if` равно нулю, то функция не возвращается и выполняет оставшиеся операторы ❹.

Вторая разновидность оператора `if` содержит предложение `else`; если не был выбран исходный оператор, то он выбирает для выполнения альтернативный:

```
if (выражение)
  подоператор1
else
  подоператор2
```

В этом формате *подоператор1* выполняется, если *выражение* не равно нулю; в противном случае выполняется *подоператор2*. В любом случае будет выполнено только одно из этих выражений.

В обоих вариантах условно выполняемый оператор может быть еще одним экземпляром `if`. Этим часто пользуются для создания лесенки из операторов `if...else`, как показано в листинге 5.3.

Листинг 5.3. Лесенка if...else

```
if (выр1)
  подоператор1
else if (выр2)
  подоператор2
else if (выр2)
  подоператор3
else
  подоператор4
```

Из четырех операторов в лесенке `if...else` будет выполнен один (и только один):

- *подоператор1* выполняется, если *выр1* не равно 0;
- *подоператор2* выполняется, если *выр1* равно 0, а *выр2* не равно 0;
- *подоператор3* выполняется, если *выр1* и *выр2* равны 0, а *выр3* не равно 0;
- *подоператор4* выполняется, только если все предыдущие условия равны 0.

Пример, представленный в листинге 5.4, использует лесенку `if...else` для вывода отметок.

Листинг 5.4. Использование лесенки if...else для вывода отметок

```
void printgrade(unsigned int marks) {
  if (marks >= 90) {
    puts("YOUR GRADE : A");
  } else if (marks >= 80) {
    puts("YOUR GRADE : B");
  } else if (marks >= 70) {
    puts("YOUR GRADE : C");
  } else {
    puts("YOUR GRADE : Failed");
  }
}
```

В этой лесенке `if...else` функция `printgrade` проверяет значение параметра `marks` типа `unsigned int`, чтобы определить, больше ли он либо равен 90. Если да, то функция выводит `YOUR GRADE : A`. В противном случае она проверяет `marks`, больше ли он или равен 80, и т. д., спускаясь по лесенке `if...else`. Если `marks` не равен и не больше 70, то функция выводит `YOUR GRADE : Failed`. В этом примере используется стиль оформления кода, в котором закрывающая фигурная скобка находится в одной строчке с предложением `else`.

Из всех операторов, которые идут за `if`, выполняется лишь один. Например, в следующем фрагменте кода `conditionally_executed_function` выполняется, только если `condition` не равно 0, а `unconditionally_executed_function` выполняется всегда:

```
if (condition)
  conditionally_executed_function();
unconditionally_executed_function(); // всегда выполняется
```

Попытки добавить еще одну условно выполняемую функцию являются распространенной причиной ошибок:

```
if (condition)
  conditionally_executed_function();
  second_conditionally_executed_function(); // ????
unconditionally_executed_function(); // всегда выполняется
```

В этом примере функция `second_conditionally_executed_function` выполняется *вне зависимости от условия*. Ее название и форматирование могут ввести в заблуждение, поскольку пробельные символы в целом и отступы в частности не имеют никакого синтаксического значения. Данный код можно исправить за счет дополнительных фигурных скобок для выделения единого составного оператора или блока, который будет выполнен как единое целое:

```
if (condition) {
  conditionally_executed_function();
  second_conditionally_executed_function(); // исправлено
}
unconditionally_executed_function(); // всегда выполняется
```

Изначальный фрагмент кода не содержал ошибок как таковых, однако во многих руководствах по оформлению кода рекомендуется всегда указывать фигурные скобки, чтобы избежать подобных недоразумений:

```
if (condition) {
  conditionally_executed_function();
}
unconditionally_executed_function(); // всегда выполняется
```

Я опускаю скобки только в случае, когда условно выполняемый оператор можно разместить в одной строчке с `if`:

```
if (!quotient) return false;
```

Данная проблема становится менее острой, если позволить интегрированной среде разработки автоматически форматировать ваш код, поскольку ей

все равно, указали вы фигурные скобки или нет. Некоторые компиляторы тоже проверяют отступы и выдают предупреждения, если те не соответствуют управляющей логике. Например, в GCC для этого предусмотрен флаг `-Wmisleading-indentation`.

Оператор switch

Оператор `switch` работает точно так же, как лесенка `if...else`, только управляющее выражение должно иметь целочисленный тип. Например, оператор `switch` в листинге 5.5 делает то же самое, что и лесенка `if...else` в листинге 5.4, при условии, что `marks` — это целое число в диапазоне от 0 до 100. Если `marks` больше 10, то будет выведена последняя строка (`YOUR GRADE : Failed`), поскольку результат деления окажется больше 10 и его перехватит предложение `default`.

Листинг 5.5. Использование оператора switch для вывода отметок

```
switch (marks/10) {
  case 10:
  case 9:
    puts("YOUR GRADE : A");
    break;
  case 8:
    puts("YOUR GRADE : B");
    break;
  case 7:
    puts("YOUR GRADE : C");
    break;
  default:
    puts("YOUR GRADE : Failed");
}
```

Оператор `switch` заставляет поток выполнения перейти к одному из трех вложенных операторов в зависимости от значения управляющего выражения и константного выражения в каждой метке `case`. После перехода код продолжает выполняться последовательно, пока не будет достигнут следующий участок с управляющей логикой. В нашем примере переход к `case 10` (который не определен) приведет к выполнению следующего оператора в `case 9`. Это продиктовано логикой программы, поскольку наивысший балл, 100, должен соответствовать отметке A, а не F.

Выполнение `switch` можно прервать, в результате чего управление перейдет к оператору, который идет сразу за `switch`. Позже в этой главе мы рас-

смотрим оператор `break` более подробно. Не забудьте указать его перед следующей меткой `case`, иначе управляющая логика перейдет к следующему предложению `switch`, что является распространенной причиной ошибок. Поскольку оператор `break` необязателен, во время компиляции его отсутствие обычно не приводит к выводу диагностических сообщений. GCC сгенерирует соответствующее предупреждение, если воспользоваться флагом `-Wimplicit-fallthrough`. В стандарте C2x ожидается появление атрибута `[[fallthrough]]`, с помощью которого программист сможет указать, что переход к следующей метке соответствует его ожиданиям; в отсутствие этого атрибута пропуск оператора `break` будет считаться случайным.

В управляющем выражении повышается разрядность целочисленного типа. Константное выражение в каждой метке `case` приводится к расширенному типу управляющего. Если преобразованное значение совпадает со значением расширенного управляющего выражения, то происходит переход к оператору, который следует за соответствующей меткой `case`. В противном случае если указана метка `default`, то управление переходит к ее оператору. Если ни одно константное выражение не совпадает с управляющим и при этом отсутствует метка `default`, то ни одно из предложений тела `switch` не выполняется. Если операторы `switch` вложенные, то метки `case` и `default` доступны только внутри того из них, который находится ближе всего.

Существуют некоторые рекомендации по использованию операторов `switch`. В листинге 5.6 показана *неправильная* реализация `switch`, в которой процентная ставка назначается банковскому счету в зависимости от его типа. Банк предлагает ограниченный набор типов счетов, поэтому они представлены в виде перечисления `AccountType`.

Листинг 5.6. Оператор switch без метки default

```
typedef enum { Savings, Checking, MoneyMarket } AccountType;
void assignInterestRate(AccountType account) {
  double interest_rate;
  switch (account) {
    case Savings:
      interest_rate = 3.0;
      break;
    case Checking:
      interest_rate = 1.0;
      break;
```

```
    case MoneyMarket:
      interest_rate = 4.5;
      break;
  }
  printf(„Interest rate = %g.\n", interest_rate);

}
```

Функция `assignInterestRate` принимает единственный параметр перечисляемого типа `assignInterestRate` и выбирает на его основе подходящую процентную ставку, которая соответствует тому или иному типу счета. Этот код в его текущем виде не содержит никаких ошибок, но если программистам захочется внести какие-либо изменения, то им придется обновлять его как минимум в двух разных местах. Предположим, в банке появился новый тип счета: депозитный сертификат. Программисты обновляют `AccountType` следующим образом:

```
typedef enum { Savings, Checking, MoneyMarket, CD } AccountType;
```

Но они забыли отредактировать оператор `switch` в функции `assignInterestRate`. Следовательно, `interest_rate` не присваивается и, когда функция пытается вывести данное значение, происходит чтение неинициализированного участка памяти. Это распространенная проблема, поскольку перечисление может быть объявлено далеко от оператора `switch`, при этом таких операторов в программе может быть несколько и все они могут использовать в своем управляющем выражении объект типа `AccountType`. Clang и GCC поддерживают флаг `-Wswitch-enum`, позволяющий диагностировать такие несоответствия на этапе компиляции. Кроме того, можно уберечься от подобных ошибок, добавив в оператор `switch` метку `default`:

```
default: abort();
```

Функция `abort` (объявленная в заголовочном файле `<stdlib.h>`) приводит к преждевременному завершению программы, вследствие чего упрощается обнаружение ошибок. Чтобы упростить тестирование этого кода, в оператор `switch` можно добавить предложение `default`, как показано в листинге 5.7.

Листинг 5.7. Оператор switch с меткой default

```
typedef enum { Savings, Checking, MoneyMarket, CD } AccountType;
void assignInterestRate(AccountType account) {
  double interest_rate;
  switch (account) {
    case Savings:
```

```
      interest_rate = 3.0;
      break;
    case Checking:
      interest_rate = 1.0;
      break;
    case MoneyMarket:
      interest_rate = 4.5;
      break;
    case CD:
      interest_rate = 7.5;
      break;
    default: abort();
  }
  printf("Interest rate = %g.\n", interest_rate);
  return;
}
```

Теперь оператор `switch` содержит предложение для `CD`, а раздел `default` не используется. Тем не менее предложение `default` лучше оставить на случай, если в дальнейшем будет добавлен еще один тип счетов.

У предложения `default` есть и недостаток: оно подавляет предупреждения со стороны компилятора и не позволяет обнаружить проблему, пока программа не будет запущена. В связи с этим лучше использовать предупреждения (если они поддерживаются вашим компилятором).

Операторы итерирования

Операторы итерирования позволяют выполнять вложенные (или составные) операторы любое количество раз (начиная с нуля) с учетом критериев завершения. Слово *«итерация»* происходит от лат. *iteratio* — «повторение». Вместо «операторов итерирования» обычно используется менее формальный термин — «циклы». *Цикл* — это «процесс, окончание которого перетекает в его начало».

Оператор while

Оператор `while` делает так, что тело цикла выполняется до тех пор, пока управляющее выражение не станет равным 0. Вычисление контрольного выражения происходит перед каждым выполнением тела цикла. Рассмотрим такой пример:

```
void f(unsigned int x) {
  while (x > 0) {
```

```
    printf("%d\n," x);
    --x;
  }
  return;
}
```

Если переменная `x` изначально не больше 0, то цикл `while` прекращается еще до выполнения своего тела. Если `x` больше 0, то ее значение выводится и декрементируется. По окончании тела цикла управляющее выражение опять проверяется. Данный процесс продолжает повторяться, пока выражение не окажется равным 0. В целом этот цикл проводит отсчет от `x` до 1.

Оператор `while` — это простой цикл входного управления, который работает, пока выполняется его входное условие. В листинге 5.8 показана реализация функции `memset` из стандартной библиотеки C. Она копирует значение `val` (приведенное к типу `unsigned char`) в первые `n` символов объекта, на который указывает `dest`.

Листинг 5.8. Функция memset из стандартной библиотеки C

```
void *memset(void *dest, int val, size_t n) {
  unsigned char *ptr = (unsigned char*)dest;
  while (n-- > 0)
    *ptr++ = (unsigned char)val;
  return dest;
}
```

Первая строчка функции `memset` приводит указатель `dest` к типу `unsigned char` и присваивает полученное значение указателю `ptr` того же типа. Это позволяет нам сохранить значение `dest`, чтобы его можно было вернуть в последней строчке функции. В оставшихся двух строчках формируется цикл `while`, который копирует значение `val` (приведенное к `unsigned char`) в каждый из первых `n` символов объекта, на который указывает `dest`. Управляющее выражение цикла `while` проверяет, действительно ли `n-- > 0`.

Аргумент `n` служит *счетчиком цикла* и декрементируется на каждой его итерации в качестве побочного эффекта вычисления управляющего выражения. В данном случае счетчик цикла уменьшается с одинаковыми интервалами, пока не достигнет минимального значения (0). Цикл выполняет `n` повторений, при этом `n` меньше или равно *границе* участка памяти, на который указывает `ptr`.

Указатель `ptr` обозначает последовательность объектов типа `unsigned char` от `ptr` до `ptr + n - 1`. Значение `val` приводится к `unsigned char` и по очереди

записывается в каждый объект. Если `n` выходит за границу объекта, на который указывает `ptr`, то цикл `while` выполняет запись за пределы данного объекта. Это неопределенное поведение и распространенный дефект безопасности, известный как *переполнение буфера*. В случае выполнения этих предварительных условий цикл `while` будет прерван без неопределенного поведения. В последней итерации выражение `n-- > 0` возвращает `0`, что приводит к окончанию цикла.

Можно написать *бесконечный цикл*, который никогда не заканчивается. Чтобы не сделать этого по случайности, перед началом цикла `while` обязательно инициализируйте все объекты, фигурирующие в управляющем выражении. Кроме того, убедитесь, что управляющее выражение будет меняться так, что цикл завершится после выполнения нужного вам количества итераций.

Оператор do...while

Оператор `do...while` похож на `while`, только вычисление управляющего выражения происходит *после* каждого выполнения тела цикла, а не перед. Благодаря этому перед проверкой условия тело цикла гарантированно выполняется один раз. Оператор итерирования `do...while` имеет следующий синтаксис:

```
do
  оператор
while ( выражение );
```

В `do...while` *оператор* выполняется всегда, после чего вычисляется *выражение*. Если *выражение* не равно 0, то поток выполнения возвращается в начало цикла и *оператор* выполняется еще раз. В противном случае управление переходит к оператору, который идет за циклом.

Оператор итерирования `do...while` часто используется для ввода/вывода, когда из потока имеет смысл прочитать еще до проверки его состояния. Это показано в листинге 5.9.

Листинг 5.9. Многократное чтение из stdin величины, единицы измерения и названия элемента

```
#include <stdio.h>
//---snip---
int count; float quant; char units[21], item[21];
```

```
do {
  count = fscanf(stdin, "%f%20s of %20s", &quant, units, item);
  fscanf(stdin,"%*[^\n]");
} while (!feof(stdin) && !ferror(stdin));
```

Этот код принимает величину с плавающей запятой, единицу измерения (в виде строки) и название элемента (тоже в виде строки) из стандартного потока ввода `stdin`, пока не будет установлен флаг конца файла или не возникнет ошибка. Ввод/вывод подробно обсуждается в главе 8.

Оператор for

Оператор `for`, вероятно, самая характерная черта языка C. Он позволяет многократно выполнять какую-то операцию и обычно используется в случаях, когда количество итераций известно еще до входа в цикл. Синтаксис оператора `for` выглядит следующим образом:

```
for (предложение1; выражение2; выражение3)
  оператор
```

`выражение2` является управляющим и вычисляется перед каждым выполнением тела цикла, а *`выражение3`* — после. Если *`предложение1`* имеет вид объявления, то область видимости любых объявляемых в нем идентификаторов охватывает оставшуюся его часть и весь цикл, включая два других выражения.

Назначение *`предложения1`*, *`выражения2`* и *`выражения3`* становится очевидным, если преобразовать оператор `for` в эквивалентный цикл `while`, как показано на рис. 5.1.

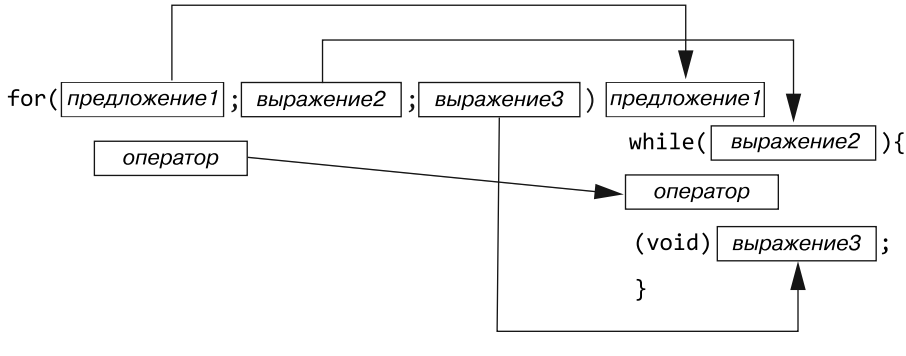

Рис. 5.1. Преобразование for в цикл while

В листинге 5.10 представлена измененная версия реализации `memset` из листинга 5.8; мы заменили `while` циклом `for`.

Листинг 5.10. Заполнение массива символов с помощью цикла for

```
void *memset(void *dest, int val, size_t n) {
  unsigned char *ptr = (unsigned char *)dest;
  for (size_t i = 0;❶ i < n;❷ ++i❸) {
    *(ptr + i) = val;
  }
  return dest;
}
```

Цикл `for` пользуется популярностью среди программистов на языке C, поскольку предоставляет удобное место для объявления и/или инициализации счетчика цикла ❶, описания управляющего выражения ❷ и инкрементирования счетчика ❸ — все это в одной строчке.

Но цикл `for` может быть несколько обманчивым. Рассмотрим в качестве примера односвязный список со структурой `node`, состоящей из элемента `data` и указателя на следующий узел в списке (`next`). Здесь также определяется указатель на структуру `node`:

```
struct node {
  int data;
  struct node *next;
};
struct node *p;
```

Используя определение `p`, следующий фрагмент кода (предназначенный для освобождения памяти, занимаемой связным списком) по ошибке считывает значение `p` после того, как оно было освобождено:

```
for (p = head; p != NULL; p = p->next)
  free(p);
```

Чтение переменной `p` после ее освобождения приводит к неопределенному поведению.

Если переписать данный список с помощью цикла `while`, то эта проблема становится очевидной:

```
p = head;
while (p != NULL) {
  free(p);
  p = p->next;
}
```

Цикл `for` может вводить в заблуждение, поскольку *выражение3* вычисляется после основного тела цикла, хотя с лексической точки зрения оно находится перед ним.

Правильный подход к выполнению этой операции состоит в сохранении необходимого указателя перед его освобождением, как показано ниже:

```
for (p = head; p != NULL; p = q) {
  q = p->next;
  free(p);
}
```

Более подробную информацию об управлении динамической памятью можно прочесть в главе 6.

Операторы перехода

Оператор перехода передает управление другому участку функции, внутри которой он объявлен, без учета каких-либо условий. Это самые низкоуровневые средства управления потоком выполнения, которые обычно четко соответствуют ассемблерному коду.

Оператор goto

Перед любым оператором можно указать *метку*, которая имеет вид идентификатора, заканчивающегося двоеточием. Оператор `goto` инициирует переход к оператору, перед которым указана заданная метка (в рамках одной и той же функции). Переход является безусловным; это значит, он производится при каждом выполнении `goto`. Ниже представлен пример оператора `goto`:

```
  /* выполняемые операторы */
  goto метка;
  /* пропущенные операторы */
метка:
  /* выполняемые операторы */
```

Выполнение продолжается, пока не будет достигнут оператор `goto`, после чего управление переходит к оператору, который идет за *меткой*. Код, который находится между `goto` и *меткой*, пропускается.

С тех пор как Эдсгер Дейкстра в 1968 году опубликовал заметку под названием Go To Statement Considered Harmful (о вреде оператора `goto`), этот

оператор имеет плохую репутацию. Критика Дейкстры заключалась в том, что несистематичное использование оператора может сделать код *похожим на спагетти* — то есть усложнить и запутать его управляющую логику, в результате чего поток выполнения программы начинает изгибаться и переплетаться. Но операторы `goto` могут сделать код и более удобочитаемым, если использовать их четким и систематичным образом.

Операторы `goto` могут принести пользу, если объединить их в цепочку для освобождения выделенных ресурсов (таких как динамически выделяемая память или открытый файл) в ситуации, когда произошла ошибка и вы должны покинуть функцию. Это происходит в связи с тем, что любое выделение ресурсов может завершиться неудачно, и во избежание утечки необходимо освободить то, что уже было выделено. Если не удается выделить первый ресурс, то такая очистка не требуется, поскольку нам нечего освобождать. Но если это произойдет со вторым ресурсом, то первый нужно освободить. Аналогично если проваливается выделение третьего ресурса, то освобождения требуют второй и первый и т. д. Эти действия приводят к дублированию и усложнению кода, предназначенного для очистки, что может быть чревато ошибками.

Одно из решений состоит в использовании вложенных операторв `if`, но если степень вложенности станет слишком высокой, то код будет сложно читать. Вместо этого для освобождения ресурсов можно воспользоваться цепочкой из операторов `goto`, как показано в листинге 5.11.

Листинг 5.11. Объединение операторов goto в цепочку для освобождения ресурсов

```
int do_something(void) {
  FILE *file1, *file2;
  object_t *obj;
  int ret_val = 0; // изначально ожидаем возвращения успешного значения

  file1 = fopen("a_file", "w");
  if (file1 == NULL) {
    ret_val = -1;
    goto FAIL_FILE1;
  }
  file2 = fopen("another_file", "w");
  if (file2 == NULL) {
    ret_val = -1;
    goto FAIL_FILE2;
  }
  obj = malloc(sizeof(object_t));
  if (obj == NULL) {
```

```
    ret_val = -1;
    goto FAIL_OBJ;
  }
  // Работаем с выделенными ресурсами

  // Освобождаем все ресурсы
  free(obj);
FAIL_OBJ: // В противном случае закрываем только то, что мы открывали
  fclose(file2);
FAIL_FILE2:
  fclose(file1);
FAIL_FILE1:
  return ret_val;
}
```

Этот код имеет простой принцип работы: ресурсы выделяются по очереди, задействуются и затем освобождаются в обратном порядке («последним пришел — первым ушел»). Если во время выделения произойдет ошибка, то программа использует `goto` для перехода на подходящий участок кода и освобождает только те ресурсы, которые были выделены.

Если использовать операторы `goto` таким систематичным образом, то код будет проще читать. В качестве реального примера можно привести функцию `copy_process` из `kernel/fork.c` в ядре Linux, которая задействует 17 меток `goto` для выполнения очистки в случае сбоя внутреннего кода.

Оператор continue

Чтобы перейти в конец тела цикла и пропустить выполнение любого кода, который остается в текущей итерации, можно использовать оператор `continue`. Например, в каждом из циклов, представленных в листинге 5.12, оператор `continue` эквивалентен `goto END_LOOP_BODY;`.

Листинг 5.12. Использование оператора continue

```
while (/* … */) {
  //---snip---
  continue;
  //---snip---
END_LOOP_BODY: ;
}
```

```
do {
  //---snip---
  continue;
  //---snip---
END_LOOP_BODY: ;
} while (/* … */);
```

```
for (/* … */) {
  //---snip---
  continue;
  //---snip---
END_LOOP_BODY: ;
}
```

Оператор `continue` зачастую используется в сочетании с условными операторами, позволяя продолжить обработку в следующей итерации цикла после достижения цели, стоявшей перед текущей итерацией.

Оператор break

Оператор `break` прерывает выполнение `switch` или оператора итерирования. Ранее в этой главе мы уже использовали `break` внутри `switch`. Оператор `break` также может завершить цикл и передать управление следующему за ним оператору. Например, цикл `for` в представленном ниже примере завершается только при нажатии на клавиатуре клавиши `Q` (в верхнем или нижнем регистре):

```
#include <stdio.h>
int main(void) {
  char c;
  for(;;) {
    puts("Press any key, Q to quit: ");
    c = toupper(getchar());
    if (c == 'Q') break;
  }
} // Цикл завершается лишь при нажатии клавиши Q в верхнем
  // или нижнем регистре
```

Оператор `break`, как правило, используется для прекращения работы цикла в случае, если тот выполнил поставленную перед ним задачу. Например, оператор `break` в листинге 5.13 завершает цикл после того, как найдется заданный ключ `key` в массиве. Если предположить, что все ключи в `arr` уникальные, то функция `find_element` будет вести себя точно так же и без оператора `break`, но, в зависимости от длины массива и местонахождения ключа, это может отрицательно сказаться на производительности, поскольку ей придется выполнить намного больше итераций.

Листинг 5.13. Принудительный выход из цикла

```
size_t find_element(size_t len, int arr[len], int key) {
  size_t pos = (size_t)-1;
  // обходим arr и ищем key
  for (size_t i = 0; i < len; ++i) {
    if (arr[i] == key) {
      pos = i;
      break; // выход из цикла
    }
  }
  return pos;
}
```

Поскольку операторы `continue` и `break` пропускают часть тела цикла, их нужно использовать с осторожностью: код, который идет за ними, не выполняется.

Оператор return

Оператор `return` прерывает выполнение текущей функции и возвращает управление тому коду, который ее вызвал. Вы уже видели в данной книге множество примеров использования `return`. Количество этих операторов в функции может быть каким угодно (начиная с нуля).

Оператор `return` может просто выйти из функции, а может вернуть выражение. Первый случай относится к функциям типа `void` (которые не возвращают значение). Если функция что-то возвращает, то оператор `return` должен вернуть выражение со значением возвращаемого типа. При этом значение выражения возвращается вызывающей стороне в качестве результата вызова функции:

```
int sum(int x, int y, int z) {
  return x + y + z;
}
```

Эта простая функция плюсует свои параметры и возвращает полученную сумму. Возвращаемое выражение `x + y + z` имеет значение типа `int`, которое соответствует возвращаемому типу функции. Если бы данное выражение имело другой тип, то было бы автоматически преобразовано в объект того же типа, который возвращает функция. Возвращать можно также элементарные выражения наподобие 0 или 1. Впоследствии результат работы функции может быть использован в другом выражении или присвоен переменной.

Имейте в виду: если управляющая логика дойдет до закрывающей скобки функции, в объявлении которой указано возвращаемое значение, и не выполнит оператор `return` с каким-то выражением, то результат работы функции будет неопределенным. Например, следующей функции не удается вернуть значение, когда переменная `a` неотрицательная, поскольку условие `a < 0` в этом случае оказывается ложным:

```
int absolute_value(int a) {
  if (a < 0) {
    return -a;
  }
}
```

Данный дефект можно легко исправить, указав возвращаемое значение в ситуации, когда переменная `a` не является отрицательной. Это показано в листинге 5.14.

Листинг 5.14. Функция absolute_value возвращает значение a в любом случае

```
int absolute_value(int a) {
  if (a < 0) {
    return -a;
  }
  return a;
}
```

Но в этом коде по-прежнему есть ошибка, если используется представление в дополнительном коде (см. главу 3). Пусть ее поиск будет для вас небольшим упражнением.

Упражнения

Попробуйте самостоятельно выполнить эти упражнения.

1. Отредактируйте функцию из листинга 5.11 так, чтобы вызывающей стороне было понятно, какой файл нельзя открыть.
2. Измените функцию `find_element` из листинга 5.13 так, чтобы она возвращала позицию ключа в `a`. Не забудьте вернуть идентификатор ошибки, если ключ не найден.
3. Исправьте оставшийся дефект в функции `absolute_value` из листинга 5.14.

Резюме

В этой главе вы познакомились с операторами управляющей логики.

- Операторы выбора, такие как `if` и `switch`, позволяют выбрать один из операторов в зависимости от значения управляющего выражения.
- Операторы итерирования многократно выполняют тело цикла, пока управляющее выражение не станет равным 0.
- Операторы перехода передают управление другому участку кода вне зависимости от каких-либо условий.

В следующей главе будет рассмотрена динамически выделяемая память.

6 Динамически выделяемая память

В главе 2 вы узнали, что у каждого объекта есть срок хранения, определяющий его время жизни, и что в языке C эти сроки делятся на четыре категории: автоматические, статические, потоковые и динамические. В этой главе речь пойдет о *динамически выделяемой памяти*, которая выделяется из кучи во время выполнения. Эта память подходит в ситуациях, когда точные размеры хранимых объектов становятся известны только во время выполнения программы.

Сначала будут описаны различия между автоматическим, статическим, потоковым и динамическим сроками хранения. Мы пропустим потоковый срок хранения, поскольку это относится к параллельному выполнению, которое в данной книге не рассматривается. Затем исследуем функции, позволяющие выделять и освобождать память, а также распространенные ошибки, возникающие в процессе выделения, и то, как их можно избежать. В этой главе, как и на практике, термины *«память»* и *«хранилище»* считаются взаимозаменяемыми.

Срок хранения

Объекты могут занимать место в *хранилище* — памяти с произвольным доступом (RAM), памяти с доступом только для чтения (ROM) или регистрах. Динамический срок хранения по своим свойствам существенно отличается от автоматического и статического. Для начала рассмотрим автоматический и статический сроки хранения, которые были описаны еще в главе 2.

Объекты с автоматическим сроком хранения объявляются внутри блока или в виде параметров функции. Время жизни этих объектов начинается и заканчивается вместе с выполнением блока, в котором они объявлены. Если блок вызывается рекурсивно, то каждый раз создается новый объект с отдельным местом в памяти.

Объекты, объявленные в области видимости файла, имеют статический срок хранения. Они существуют на протяжении всей работы программы, а их хранимые значения инициализируются еще до ее запуска. Вы также можете объявить переменную в области видимости блока со статическим сроком хранения, используя спецификатор класса хранения `static`.

Куча и менеджеры памяти

Динамически выделяемая память имеет *динамический срок хранения*. Время жизни динамически выделенного объекта находится в промежутке между его выделением и освобождением. Этот вид памяти выделяется из *кучи*. Она представляет собой один или несколько крупных блоков памяти, которые могут быть поделены на части и управляются менеджером памяти.

Менеджеры памяти (memory manager) — это библиотеки для управления кучей, предоставляющие реализации стандартных функций по работе с памятью, которые будут описаны ниже. Менеджер памяти выполняется в рамках клиентского процесса. Он запрашивает у операционной системы один или несколько блоков памяти и затем выделяет их для клиентского кода, когда тот вызывает функцию выделения памяти.

Менеджеры памяти работают лишь с еще не выделенной и с уже освобожденной памятью. Как только память была выделена и до тех пор, пока она не освобождена, ее управлением занимается вызывающая сторона. Именно она отвечает за ее освобождение, хотя в большинстве реализаций это происходит автоматически при завершении программы.

Реализации менеджеров памяти

Менеджеры памяти обычно реализуют один из вариантов алгоритма выделения динамической памяти, описанного Дональдом Кнутом (1997). Этот алгоритм использует *теги границ* — поля, которые находятся по обе стороны блока памяти, возвращенного программисту, и описывают его размер. Информация

о размере позволяет перебрать все блоки памяти, начиная с любого известного блока и двигаясь в любом направлении; благодаря этому менеджер памяти может объединить два смежных свободных блока в один, чтобы минимизировать фрагментацию.

Фрагментация возникает, когда выделение и освобождение памяти приводит к появлению множества мелких и нехватке крупных блоков. В таком случае выделить большой блок памяти оказывается невозможным, хотя суммарного количества свободной памяти должно было бы хватить. Память, выделяемая клиентскому процессу и предназначенная для использования внутри менеджеров, находится в рамках единого адресного пространства клиентского процесса.

Когда следует использовать динамически выделяемую память

Динамически выделяемая память используется, когда точные размеры хранимых объектов становятся известны только во время выполнения программы. Она менее эффективна по сравнению со статически выделяемой, так как менеджеру памяти приходится искать в куче блоки подходящего размера во время выполнения, а затем, когда они больше не нужны, вызывающая сторона должна явно освобождать их. Все это требует дополнительных вычислений. Объекты, размеры которых известны на этапе компиляции, по умолчанию следует объявлять с автоматическим или статическим сроком хранения.

Когда динамически выделенную память не возвращают менеджеру памяти, происходит *утечка памяти*. Если эти утечки станут значительными по объему, то менеджер памяти рано или поздно потеряет возможность выделять новые блоки. Кроме того, динамически выделяемая память требует дополнительных вычислительных ресурсов для вспомогательных операций, таких как *дефрагментация* (консолидация смежных свободных блоков). К тому же в целях обеспечения работы этих процессов менеджер памяти зачастую использует дополнительное место, где хранит свои управляющие структуры.

Динамически выделяемая память обычно используется, когда размер или количество объектов неизвестны на этапе компиляции. Например, с помощью памяти этого типа во время выполнения программы можно прочитать таблицу из файла, особенно если вы не знаете заранее, сколько

в ней строк. Похожим образом динамически выделяемую память можно использовать для создания связных списков, хеш-таблиц, двоичных деревьев и других структур данных, состоящих из элементов, количество которых неизвестно в момент компиляции.

Функции для управления памятью

Стандартная библиотека C предоставляет функции для выделения и освобождения динамической памяти. В их число входят `malloc`, `aligned_alloc`, `calloc` и `realloc`. Освобождение можно выполнить с помощью функции `free`. В OpenBSD есть функция `reallocarray`, которая не является частью стандартной библиотеки, но тоже может пригодиться для выделения памяти.

Функция malloc

Функция `malloc` выделяет место для объекта заданного размера, начальное значение которого невозможно определить. В листинге 6.1 мы вызываем функцию `malloc`, чтобы динамически выделить место для объекта размера `struct widget`.

Листинг 6.1. Выделение места для widget с помощью функции malloc

```
#include <stdlib.h>
typedef struct {
  char c[10];
  int i;
  double d;
} widget;

❶ widget *p = malloc(sizeof(widget));
❷ if (p == NULL) {
    // Обрабатываем ошибку выделения
  }
  // Продолжаем обработку
```

Все функции для выделения памяти принимают аргумент типа `size_t`, определяющий количество байтов, которые нужно выделить ❶. В целях переносимости при вычислении размера объектов мы используем операцию `sizeof`, поскольку в разных реализациях различные типы, такие как `int` и `long`, могут иметь разную разрядность.

Функция `malloc` возвращает либо нулевой указатель, чтобы сообщить об ошибке, либо указатель на выделенный участок. В связи с этим мы проверяем, возвращает ли `malloc` нулевой указатель ❷, и обрабатываем ошибку соответствующим образом.

После того как функция успешно вернет выделенный участок, мы можем обращаться к членам структуры `widget`, используя указатель `p`. Например, `p->i` позволяет обратиться к члену `widget` типа `int`, а `p->d` предоставляет доступ к члену типа `double`.

Выделение памяти без объявления типа

Значение, которое возвращает `malloc`, можно хранить в виде указателя на `void`, чтобы не объявлять тип объекта, на который тот ссылается:

```
void *p = malloc(size);
```

Кроме того, вы можете использовать указатель на `char`, как это было принято делать до появления в C типа `void`:

```
char *p = malloc(size);
```

В обоих случаях у объекта, на который указывает `p`, нет типа, пока он не скопирован в выделенный участок. Когда это происходит, объекту назначается *фактический тип* последнего объекта, скопированного в данный блок памяти. В следующем примере блок, на который указывает `p`, получает фактический тип `widget` после вызова `memcpy`.

```
widget w = {"abc", 9, 3.2};
memcpy(p, &w, sizeof(widget));     // приведено к указателям void *
printf("p.i = %d.\n", p->i);
```

Поскольку в выделенном блоке можно хранить объекты любых типов, мы можем направлять указатели, возвращаемые любыми функциями выделения памяти, включая `malloc`, на любой тип объектов. Например, если реализация поддерживает объекты с 1-, 2-, 4-, 8- и 16-байтными выравниваниями, то при выделении 16 или больше байт памяти возвращаемый указатель будет иметь выравнивание, кратное 16.

Приведение указателя к типу объявленного объекта

Даже опытные программисты на C имеют разные мнения о том, нужно ли приводить указатель, возвращенный функцией `malloc`, к типу объявлен-

ного объекта. Следующий оператор присваивания приводит указатель к типу `widget *`:

```
widget *p = (widget *)malloc(sizeof(widget));
```

Строго говоря, это приведение типов не является обязательным. Язык C позволяет косвенно преобразовать указатель типа `void` (который вернула функция `malloc`) в указатель на объект любого типа с подходящим выравниванием (в противном случае поведение неопределено). Ручное приведение результата `malloc` к нужному типу дает возможность компилятору обнаружить непреднамеренные преобразования указателей, а также несоответствие размеров выделяемого блока и типа объекта указателя в выражении приведения.

В примерах, приводимых в данной книге, обычно используется ручное приведение типов, но оба стиля являются приемлемыми. Более подробную информацию об этом можно найти в правиле MEM02-C стандарта CERT C (немедленно приводите результат вызова функции для выделения памяти к указателю на выделенный тип).

Чтение неинициализированной памяти

Содержимое памяти, возвращенной из `malloc`, *не инициализировано*. Это значит, в нем находятся неопределенные значения. Чтение неинициализированной памяти всегда плохая идея, и такую операцию следует считать неопределенным поведением. Если хотите узнать больше, то рекомендую обратиться к моей подробной статье о *неинициализированном чтении* (Сикорд, 2017). Функция `malloc` не инициализирует возвращаемую память, поскольку ожидается, что вы все равно ее перезапишете.

Тем не менее многие новички ошибочно предполагают, что память, возвращенная из `malloc`, содержит нули. Программа, представленная в листинге 6.2, допускает именно эту ошибку.

Листинг 6.2. Ошибка инициализации

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


int main(void) {
  char *str = (char *)malloc(16);
```

```
  if (str) {
    strncpy(str, "123456789abcdef", 15);
    printf("str = %s.\n", str);
    free(str);
    return EXIT_SUCCESS;
  }
  return EXIT_FAILURE;
}
```

Эта программа выделяет 16 байт памяти, вызывая `malloc`, и затем использует `strncpy`, чтобы скопировать первые 15 байт строки в выделенный блок. Программист пытается создать корректную строку с нуль-символом в конце, копируя на один байт меньше, чем размер выделенной памяти. Делая это, программист предполагает, что в выделенном блоке уже содержится значение 0, которое будет служить нулевым байтом. Но вместо этого там вполне может находиться ненулевое значение, и в этом случае строка не будет оформлена как следует, а вызов `printf` приведет к неопределенному поведению.

Распространенное решение этой проблемы состоит в том, чтобы записать нуль-символ в последний байт выделенного блока, как показано ниже:

```
strncpy(str, "123456789abcdef", 15);
str[15] = '\0';
```

Если исходная строка меньше 15 байт, то будет скопирован нуль-символ и присваивание в строчке ❶ окажется лишним. Если же исходная строка занимает 15 или больше байт, то наличие этого присваивания будет гарантировать, что она заканчивается нуль-символом.

Функция aligned_alloc

Функция `aligned_alloc` похожа на `malloc`, но вместе с размером выделяемого объекта требует также указать его выравнивание. Ниже представлена ее сигнатура, где `size` определяет размер объекта, а `alignment` — выравнивание:

```
void *aligned_alloc(size_t alignment, size_t size);
```

Появление функции `aligned_alloc` в стандарте C11 объясняется тем, что некоторое аппаратное обеспечение имеет более строгие требования к выравниванию памяти. И хотя язык C требует, чтобы память, динамически выделяемая из `malloc`, была достаточно выровнена для всех стандартных типов, включая массивы и структуры, иногда может возникнуть необхо-

димость в переопределении тех решений, которые компилятор принимает по умолчанию.

Функция `aligned_alloc`, как правило, используется с целью задать более строгое выравнивание по сравнению со стандартным (то есть с большей степенью двойки). Если значение `alignment` не является корректным выравниванием, которое поддерживается реализацией, то данная функция завершается неудачно и возвращает нулевой указатель. Больше о выравнивании можно узнать в главе 2.

Функция calloc

Функция `calloc` выделяет место для массива с `nmemb` объектами, каждый из которых занимает `size` байт. Ее сигнатура выглядит так:

```
void *calloc(size_t nmemb, size_t size);
```

Данная функция заполняет блок памяти нулевыми байтами. Они могут отличаться от значения, которое используется для представления нуля в типах с плавающей запятой или константных нулевых указателей. Функция `calloc` также позволяет выделить место и для одиночного объекта, который в этом случае можно считать массивом, состоящим из одного элемента.

Внутри функция `calloc` умножает `nmemb` на `size`, чтобы определить количество байтов, которое нужно выделить. По историческим причинам некоторые реализации `calloc` не проверяли, переполняются ли эти значения при умножении. Однако современные версии `calloc` выполняют данную проверку, и если произведение нельзя представить с помощью типа `size_t`, то возвращают нулевой указатель.

Функция realloc

Функция `realloc` увеличивает или уменьшает размер ранее выделенного блока памяти. Она принимает указатель на некий блок, выделенный одним из предыдущих вызовов `aligned_alloc`, `malloc`, `calloc` или `realloc` (или нулевой указатель) и размер. Ее сигнатура выглядит так:

```
void *realloc(void *ptr, size_t size);
```

Вы можете использовать функцию `realloc` для расширения или (что случается реже) сужения массива.

Борьба с утечками памяти

Чтобы не наделать ошибок при использовании `realloc`, вы должны понимать (на концептуальном уровне), как реализована эта функция. Обычно она вызывает `malloc` для выделения нового участка памяти, а затем копирует в него содержимое старого участка так, чтобы не превысить ни старый, ни новый размер. Если новый участок больше старого, то функция `realloc` оставляет лишнее место неинициализированным. Если ей удается выделить новый объект, то она вызывает `free`, чтобы освободить старый. В случае неудачи `realloc` сохраняет данные старого объекта по тому же адресу и возвращает нулевой указатель. Причиной неудачного вызова `realloc` может быть, к примеру, нехватка памяти, доступной для выделения запрошенного количества байтов. Следующий пример использования `realloc` содержит ошибку:

```
size += 50;
if ((p = realloc(p, size)) == NULL) return NULL;
```

В этом примере переменная `size` инкрементируется на 50, после чего вызывается `realloc`, чтобы увеличить размер блока, на который указывает `p`. Если вызов `realloc` завершается неудачно, то указателю `p` присваивается значение `NULL`, однако блок, на который указывает `p`, не освобождается, что приводит к утечке памяти.

В листинге 6.3 показано, как правильно использовать функцию `realloc`.

Листинг 6.3. Правильное использование функции realloc

```
void *p2;
void *p = malloc(100);
//---snip---
if ((nsize == 0) || (p2 = realloc(p, nsize)) == NULL) {
  free(p);
  return NULL;
}
p = p2;
```

В данном фрагменте кода объявляются две переменные, `p` и `p2`. Первая указывает на динамически выделенную память, которую вернула функция `malloc`, а вторая изначально не инициализируется. В конечном счете мы меняем размер этой памяти, вызывая функцию `realloc` с указателем на `p` и новым размером `nsize`. Значение, возвращаемое из `realloc`, присваивается переменной `p2`, чтобы не перезаписывать указатель, хранящийся в `p`. Если `realloc` возвращает нулевой указатель, то память, на которую указывает `p`, освобождается и функция возвращает `NULL`. Если все про-

шло хорошо, и realloc возвращает указатель на блок размером nsize, то переменной p присваивается указатель на вновь выделенный блок и выполнение продолжается.

Данный код также включает проверку на выделение нуля байт. Функции realloc не следует передавать 0 в качестве аргумента size, поскольку это фактически (а в C2x официально) является неопределенным поведением.

Следующий вызов функции realloc не возвращает нулевой указатель, однако адрес, который хранится в p, становится недействительным и читать его недопустимо:

```
newp = realloc(p, ...);
```

В частности, следующая проверка является недопустимой:

```
if (newp != p) {
  // обновляем указатели с учетом заново выделенной памяти
}
```

После вызова realloc, вне зависимости от того, менял этот вызов адрес выделенного блока или нет, любые указатели на память, на которую ранее ссылался указатель на p, должны быть перенаправлены к блоку, на который ссылается newp.

Чтобы решить эту проблему, можно ввести дополнительную абстракцию, иногда называемую *дескриптором (handle)*. Если указатель всегда используется косвенно, то при его перенаправлении будут автоматически обновлены все участки кода, в которых он фигурирует.

Вызов realloc с нулевым указателем

Вызов realloc с нулевым указателем эквивалентен вызову malloc. При условии, что значение newsize не равно 0, следующий код:

```
if (p == NULL)
  newp = malloc(newsize);
else
  newp = realloc(p, newsize);
```

равнозначен:

```
newp = realloc(p, newsize);
```

Первая, более длинная версия кода вызывает malloc при первом выделении памяти и realloc, если позже потребуется изменить размер. Но поскольку

вызов `realloc` с нулевым указателем эквивалентен вызову `malloc`, вторая версия делает то же самое, только в сжатом виде.

Функция reallocarray

Функция `reallocarray` в OpenBSD может заново выделить память для массива и при этом проверяет на переполнение, когда вычисляет его размер. Таким образом, вам не нужно проводить данную проверку самостоятельно. Сигнатура функции `reallocarray` выглядит так:

```
void *reallocarray(void *ptr, size_t nmemb, size_t size);
```

Функция `reallocarray` выделяет память для `nmemb` размера `size` и проверяет, не происходит ли целочисленное переполнение при вычислении `nmemb * size`. Другие платформы, включая GNU C Library (libc), тоже внедрили у себя эту функцию, и теперь ее предлагают включить в следующую версию стандарта POSIX. Стоит отметить, что `reallocarray` не обнуляет выделенную память.

Как мы уже видели в предыдущих главах, целочисленное переполнение — это серьезная проблема, которая может приводить к переполнениям буферов и другим уязвимостям безопасности. Например, в следующем коде выражение `num * size` может переполниться до того, как будет передано функции `realloc` в качестве аргумента `size`:

```
if ((newp = realloc(p, num * size)) == NULL) {
  //---snip---
```

Функция `reallocarray` подходит в ситуациях, когда для определения размера выделяемого блока умножаются два значения:

```
if ((newp = reallocarray(p, num, size)) == NULL) {
  //---snip---
```

Если `num * size` грозит переполнением, то этот вызов `reallocarray` завершится неудачно и вернет нулевой указатель.

Функция free

Когда динамически выделенная память больше не нужна, ее следует освободить путем вызова функции `free`. Освобождение памяти играет важную роль, позволяя повторно задействовать одни и те же блоки. Это снижает

риск расхода всей доступной памяти и делает использование кучи более эффективным.

Для освобождения памяти нужно передать указатель на нее функции `free`, которая имеет следующую сигнатуру:

```
void free(void *ptr);
```

Значение `ptr` должно быть предварительно получено из вызова `aligned_alloc`, `malloc`, `calloc` или `realloc`. В правиле MEM34-C стандарта CERT C (вызывайте `free` только для динамически выделенной памяти) описывается, что происходит, если значение не возвращается. Память является ограниченным ресурсом, и ее необходимо восстанавливать.

Если вызвать функцию `free` с нулевым указателем, то она просто завершится и ничего не произойдет:

```
char *ptr = NULL;
free(ptr);
```

Предотвращение уязвимостей двойного освобождения

Если вызвать функцию `free` для одного и того же указателя несколько раз, то возникнет неопределенное поведение. Подобные дефекты могут приводить к дырам в безопасности, известным как *уязвимости двойного освобождения*. Ими потенциально могут воспользоваться для выполнения произвольного кода с правами уязвимого процесса. Мы не станем обсуждать здесь все возможные последствия двойного освобождения, но я подробно рассматриваю их в книге *Secure Coding in C and C++* (Сикорд, 2013). Особенно часто эти уязвимости встречаются в коде для обработки ошибок, где программисты пытаются освобождать выделенные ресурсы.

Еще одна распространенная ошибка — обращение к памяти, которую уже освободили. Она часто остается невыявленной, так как код с виду работает, а неожиданные сбои происходят не там, где ее допустили. В листинге 6.4 показан код из настоящего приложения, где аргумент `close` оказывается недействительным, поскольку блок памяти, на который раньше ссылался указатель `dirp`, был освобожден вторым вызовом `free`.

Листинг 6.4. Обращение к уже освобожденной памяти

```
#include <dirent.h>
#include <stdlib.h>
```

```
#include <unistd.h>


int closedir(DIR *dirp) {
  free(dirp->d_buf);
  free(dirp);
  return close(dirp->d_fd); // dirp уже освобождена
}
```

Указатели на уже освобожденную память называют *висячими*. Они являются потенциальным источником ошибок (как банановая кожура на полу), поскольку с их помощью можно выполнить запись в память, которую уже освободили, или передать их функции `free`, что провоцирует уязвимости двойного освобождения. Больше информации на эту тему можно найти в правиле MEM30-C стандарта CERT C (не обращайтесь к освобожденной памяти).

Обнуление указателя

Чтобы ограничить возможность возникновения дефектов с висячими указателями, после завершения вызова `free` указателю следует присвоить `NULL`:

```
char *ptr = malloc(16);
//---snip---
free(ptr);
ptr = NULL;
```

Любые последующие попытки разыменования данного указателя обычно заканчиваются аварийным завершением программы (это повышает вероятность выявления ошибки во время разработки и тестирования). Если указатель равен `NULL`, то память можно освобождать повторно без каких-либо последствий. К сожалению, функция `free` не может обнулить указатель самостоятельно, поскольку ей передается лишь его копия.

Состояния памяти

Динамически выделяемая память может находиться в одном из трех состояний, показанных на рис. 6.1: она может быть невыделенной и неинициализированной внутри менеджера памяти, выделенной, но неинициализированной, а также выделенной и инициализированной. Переход

из одного состояния в другое происходит в результате вызовов функций `malloc` и `free` или записи в память.

Рис. 6.1. Состояния памяти

В каждом состоянии памяти допустимы разные операции. Избегайте операций с памятью, которые не относятся к допустимым или явно помечены как недопустимые. Это относится к каждому байту памяти, поскольку инициализированные байты читать можно, а неинициализированные — нельзя.

Структуры с массивами произвольной длины

В языке C выделение места для структур, содержащих массивы, всегда было немного проблематичным. Если массив имеет фиксированное количество элементов, то никаких проблем не возникает, поскольку размер структуры можно легко определить. Но разработчикам зачастую приходится объявлять массивы произвольной длины (flexible array, иногда называют гибкими), и раньше язык C не предлагал для этого никаких простых решений.

В настоящее же время вы можете объявлять и выделять место для структур с любым количеством фиксированных членов, последний из которых

является массивом неизвестного размера. Начиная с C99, последний (но не единственный) член структуры может быть *массивом неполного типа*; это значит, что размер массива неизвестен. Таким образом, возможно определять размер на этапе выполнения. Структуры с массивами произвольной длины позволяют обращаться к объектам переменной длины.

Например, в листинге 6.5 демонстрируется использование массива `data` произвольной длины в структуре `widget`. Мы динамически выделяем место для объекта, вызывая функцию `malloc`.

Листинг 6.5. Структура с массивом произвольной длины

```
#include <stdlib.h>

typedef struct {
  size_t num;
❶ int data[];
} widget;

void *func(size_t array_size) {
❷ widget *p = (widget *)malloc(sizeof(widget) + sizeof(int) * array_size);
  if (p == NULL) {
    return NULL;
  }

  p->num = array_size;
  for (size_t i = 0; i < p->num; ++i) {
❸ p->data[i] = 17;
  }
}
```

Мы объявили структуру, последний член которой, массив `data` ❶, имеет неполный тип (без указания размера). Затем мы выделили память для всей структуры ❷. При вычислении ее размера с помощью операции `sizeof` массив произвольной длины игнорируется. Поэтому при выделении памяти мы должны явно указать подходящий размер для такого массива. Для этого мы выделяем ему дополнительные байты, умножая количество его элементов (`array_size`) на размер каждого из них (`sizeof(int)`). В данной программе предполагается, что при умножении `array_size` на `sizeof(int)` не происходит переполнение.

Для обращения к этой памяти можно использовать операции `.` или `->` ❸, как будто она была выделена для `data[array_size]`. Более подробную информацию о выделении и копировании структур с массивами произвольной длины

можно найти в правиле MEM33-C стандарта CERT C (выделяйте и копируйте структуры, содержащие массивы произвольной длины, динамически).

До появления C99 некоторые компиляторы использовали похожий прием, предлагая разный синтаксис. Правило DCL38-C стандарта CERT C (используйте корректный синтаксис при объявлении структур с массивами произвольной длины) служит напоминанием о том, что вы должны использовать синтаксис, доступный в C99 и последующих стандартах языка C.

Другие виды динамически выделяемой памяти

Помимо функций, которые позволяют выделять память из кучи, в языке C и стандартной библиотеке есть и другие средства динамического выделения. Они, как правило, позволяют выделять память в стековом фрейме вызывающей стороны (стек не определен в стандарте C, но является распространенным механизмом во многих реализациях). *Стек* — это структура данных вида «последним пришел — первым ушел» (last-in-first-out, LIFO), поддерживающая вложенные вызовы функций на этапе выполнения. Каждый вызов создает *стековый фрейм*, в котором могут храниться локальные переменные (с автоматическим сроком хранения) и другие данные, относящиеся к вызовам функций.

Функция alloca

По соображениям производительности в некоторых реализациях предусмотрена функция `alloca`, которая позволяет динамически выделять память в стеке, а не в куче. Данная память автоматически освобождается при возвращении из функции, которая вызвала `alloca`. Функция `alloca` является *встроенной на уровне компилятора (intrinsic)*; то есть за ее реализацию отвечает сам компилятор. Это позволяет ему подставлять вместо исходного вызова последовательность автоматически сгенерированных инструкций. Например, в архитектуре x86 компилятор заменяет вызов `alloca` единственной инструкцией, которая меняет указатель стека таким образом, чтобы выделить дополнительное место.

Функция `alloca` берет начало в ранних версиях операционной системы Unix, которые разрабатывались в Bell Labs, но не является частью

стандартной библиотеки или POSIX. В листинге 6.6 показан пример функции `printerr`, которая, прежде чем выводить строку с ошибкой в `stderr`, выделяет для нее память с помощью `alloca`.

Листинг 6.6. Функция printerr

```
void printerr(errno_t errnum) {
  rsize_t size = strerrorlen_s(errnum) + 1;
  char *msg = (char *)alloca(size);
  if (strerror_s(msg, size, errnum) != 0) {
    fputs(msg, stderr);
  }
  else {
    fputs("unknown error", stderr);
  }
}
```

Функция `printerr` принимает один аргумент, `errnum` типа `errno_t`. В ее первой строчке вызывается `strerrorlen_s`, чтобы определить длину строки, относящейся к ошибке с этим конкретным номером. Узнав размер массива, необходимый для хранения строки с ошибкой, мы можем вызвать функцию `alloca`, чтобы выделить для него место в памяти. Затем мы получаем строку с описанием ошибки, используя функцию `strerror_s`, и сохраняем результат в только что выделенный блок, на который указывает `msg`. Если `strerror_s` завершится удачно, то мы выведем сообщение об ошибке; в противном случае будет выведена строка `unknown error`. Эта функция `printerr` служит демонстрацией применения `alloca`, и в действительности ее можно было бы сделать попроще.

Использовать функцию `alloca` не так-то просто. Прежде всего, ее вызов может выделить блок, который выходит за пределы стека, но при этом она не возвращает нулевой указатель, что не дает нам обнаружить ошибку. Как следствие, крайне важно избегать использования функции `alloca` для выделения крупных или неограниченных участков памяти. В данном примере вызов `strerrorlen_s` должен вернуть адекватный размер выделенной памяти.

Функции `alloca` присуща еще одна проблема: программисты могут по ошибке сделать для нее вызов `free`. Использование вызова `free` с указателем, полученным не из `aligned_alloc`, `calloc`, `realloc` или `malloc`, является серьезной ошибкой. Кроме того, компиляторы обычно избегают встраивания функций, которые вызывают `alloca`. По этим причинам применение `alloca` не приветствуется.

Компилятор GCC предоставляет флаг `-Walloca` для диагностики вызовов `alloca` и флаг `-Walloca-larger-than=`*`size`*, диагностирующий все вызовы `alloca`, в которых размер запрашиваемой памяти превышает *`size`*.

Массивы переменной длины

Массивы переменной длины (variable-length arrays, VLA) появились в C99. При их объявлении можно указать переменную, которая определяет их размеры на этапе выполнения. После создания массива его размер нельзя изменить (вопреки их названию). VLA подходит в ситуациях, когда количество элементов массива неизвестно до выполнения программы. Все объявления VLA должны находиться в области видимости блока или прототипа функции. Примеры этих двух сценариев рассмотрены в следующих разделах.

Объявление на уровне блока

В следующей функции `func` используется автоматическая переменная *на уровне блока* для объявления массива переменной длины с размером `size`:

```
void func(size_t size) {
  int vla[size];
  //---snip---
}
```

Массив выделяется в стековом фрейме и освобождается при выходе из текущего фрейма — подобно тому как это делает функция `alloca`[1]. Ниже показана измененная версия листинга 6.6, в которой вместо вызова `alloca` используется VLA. Изменение ограничено лишь одной строчкой кода (выделенной жирным шрифтом) (листинг 6.7).

Листинг 6.7. Функция print_error, переписанная с использованием VLA

```
void print_error(int errnum) {
  size_t size = strerrorlen_s(errnum) + 1;
  char msg[size];
  if (strerror_s(msg, size, errnum) != 0) {
    fputs(msg, stderr);
  }
```

[1] На самом деле стандарт не оговаривает, как должны выделяться массивы переменной длины; некоторые компиляторы выделяют их в куче, просто неявно вызывая malloc в начале блока и free в его конце. — *Примеч. науч. ред.*

```
  else {
    fputs("unknown error", stderr);
  }
}
```

Основное преимущество VLA вместо функции `alloca` состоит в том, что данный синтаксис соответствует представлению программиста о том, как работают массивы с автоматическим сроком хранения, то есть занимаемую ими память не нужно освобождать явно.

Массивам переменной длины присущи некоторые проблемы, характерные для функции `alloca`. Например, они могут попытаться занять память, выходящую за пределы стека. К сожалению, не существует переносимых методов определения оставшегося в стеке места, поэтому обнаружить данную ошибку невозможно. К тому же вычисление размера массива может переполниться, если умножить его длину на размер каждого элемента. В связи с этим размер массива необходимо проверять до его объявления, чтобы избежать выделения участков памяти, которые слишком велики или имеют некорректный размер. Это может быть особенно важным в функциях, вызываемых рекурсивно, так как для каждого вызова создается совершенно новый набор автоматических переменных (включая и эти массивы).

Вы должны определить, хватит ли вам места в стеке в самом худшем случае (при выделении максимально крупных блоков с глубокой рекурсией). В некоторых реализациях для VLA можно указывать и отрицательный размер[1], поэтому убедитесь в том, что ваш размер представлен с помощью `size_t` или другого беззнакового типа. Более подробную информацию об этом можно найти в правиле ARR32-C стандарта CERT C (следите за тем, чтобы аргументы с размерами массивов переменной длины находились в допустимом диапазоне). В GCC можно воспользоваться флагом `-Wvla-larger-than=size` для диагностики определений VLA, которые либо превышают заданный размер, либо имеют недостаточно жесткие границы.

Наконец, еще одна интересная и, вероятно, неожиданная ситуация возникает при вызове `sizeof` для VLA. Операция `sizeof` обычно выполняется на этапе компиляции. Но если какое-то выражение изменяет размер массива, то оно будет вычислено во время выполнения со всеми своими побочными эффектами. То же самое касается и `typedef`. Это показано на примере программы из листинга 6.8.

[1] Синтаксически можно, но это приведет к неопределенному поведению согласно § 6.7.5.2.5 стандарта C99 (ISO/IEC 9899:TC3). — *Примеч. науч. ред.*

Листинг 6.8. Неожиданные побочные эффекты

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  size_t size = 12;
  printf("%zu\n", size); // выводит 12
  (void)sizeof(int[size++]);
  printf("%zu\n", size); // выводит 13
  typedef int foo[size++];
  printf("%zu\n", size); // выводит 14
}
```

В этой простой демонстрационной программе мы объявляем переменную `size` типа `size_t` и присваиваем ей `12`. Затем вызываем операцию `sizeof` с `int[size++]` в качестве аргумента. Поскольку данное выражение изменяет размер массива, переменная `size` инкрементируется и становится равной `13`. Аналогичным образом `typedef` инкрементирует ее до `14`.

Объявление на уровне прототипа функции

VLA также можно объявить в качестве параметра функции. Как вы помните из главы 2, при использовании в выражении массив преобразуется в указатель на свой первый элемент. Это значит, мы должны вручную добавить еще один параметр, чтобы указать размер массива. Например, в сигнатуре `memset` для этого предусмотрен параметр `n`:

```
void *memset(void *s, int c, size_t n);
```

При вызове такой функции параметр `n` должен точно отражать размер массива, на который указывает `s`. Если это значение больше фактического размера, то возникает неопределенное поведение.

В случае объявления функции, принимающей в качестве аргумента VLA с указанием размера, этот размер должен быть объявлен до объявления самого массива, в котором он указывается. Например, мы могли бы изменить сигнатуру функции `memset` так, чтобы она принимала VLA:

```
void *memset_vla(size_t n, char s[n], int c);
```

Здесь мы изменили порядок следования параметров таким образом, чтобы переменная `n` типа `size_t` объявлялась перед тем, как мы используем ее в объявлении массива. Аргумент массива по-прежнему сводится к указателю, и в результате этого объявления не выделяется никакая

память. Вызывая данную функцию, вы сами должны выделить место для массива, на который указывает `s`, и убедиться в том, что `n` будет для него допустимым размером.

Массивы переменной длины позволяют сделать функции более обобщенными и, следовательно, полезными. Например, функция `matrix_sum` суммирует все элементы двумерного массива. Следующая ее версия принимает матрицу с фиксированной длиной столбца:

```
int matrix_sum(size_t rows, int m[][4]);
```

При передаче в функцию многомерного массива теряется информация о размере его первого измерения, но ее все равно необходимо передать в качестве аргумента. В данном примере эта информация предоставляется параметром `row`. Как показано в листинге 6.9, вы можете вызывать данную функцию для сложения значений любой матрицы, у которой ровно четыре столбца.

Листинг 6.9. Сложение значений в матрицах с четырьмя столбцами

```
int main(void) {
  int m1[5][4];
  int m2[100][4];
  int m3[2][4];
  printf("%d.\n", matrix_sum(5, m1));
  printf("%d.\n", matrix_sum(100, m2));
  printf("%d.\n", matrix_sum(2, m3));
}
```

Все это будет хорошо работать до тех пор, пока вам не понадобится сложить значения матрицы, количество столбцов которой не равно четырем. Например, если указать для `m3` пять столбцов, то будет показано следующее предупреждение:

```
warning: incompatible pointer types passing 'int [2][5]' to parameter
of type 'int (*)[4]'
```

В таких ситуациях нужно написать новую функцию с сигнатурой, которая соответствует новым размерам многомерного массива. Но проблема этого подхода в том, что он не позволяет добиться достаточного обобщения.

Вместо этого мы можем переписать функцию `matrix_sum` так, чтобы она использовала VLA. Данное изменение, представленное в листинге 6.10, позволяет вызывать `matrix_sum` для матриц любой размерности.

Листинг 6.10. Использование VLA в качестве параметра функции

```
int matrix_sum(size_t rows, size_t cols, int m[rows][cols]) {
  int total = 0;

  for (size_t r = 0; r < rows; r++)
    for (size_t c = 0; c < cols; c++)
      total += m[r][c];
  return total;
}
```

И снова ни объявление, ни определение функции не приводит к выделению памяти. Место для матрицы необходимо выделять отдельно, а ее размеры должны совпадать с теми, которые вы передали функции в качестве аргументов `rows` и `cols`. В противном случае проведение программы будет неопределенным.

Отладка проблем, связанных с выделением памяти

Ранее в данной главе уже отмечалось, что ненадлежащее управление памятью может приводить к таким ошибкам, как утечки, чтение или запись в освобожденный блок и повторное освобождение одного и того же блока. Как мы уже видели, избежать некоторых из этих проблем можно за счет обнуления указателей после вызова `free`. Другой способ избегания этих проблем — максимально упростить стратегию управления динамической памятью. Например, память должна выделяться и освобождаться в одном и том же модуле, на одном уровне абстракции; если это делать в разных участках кода, то можно запутаться в том, когда и где происходит освобождение памяти (и происходит ли вообще).

Третий вариант состоит в использовании *средств динамического анализа*, которые выявляют ошибки работы с памятью и сообщают о них. Эти средства, а также общий подход к отладке, тестированию и анализу обсуждаются в главе 11. В данном разделе мы рассмотрим один из таких инструментов — `dmalloc`.

Библиотека dmalloc

Библиотека `dmalloc` (от debug memory allocation — отладка выделения памяти), созданная Греем Уотсоном, заменяет `malloc`, `realloc`, `calloc`, `free`

и другие средства управления памятью процедурами, предоставляющими механизмы отладки, которые можно настраивать во время выполнения. Эта библиотека работает на целом ряде платформ.

Чтобы сконфигурировать, собрать и установить `dmalloc`, следуйте инструкциям, доступным на сайте https://dmalloc.com/. В листинге 6.11 показано, как настроить эту библиотеку так, чтобы она сообщала о файлах и номерах строчек с проблемными вызовами. Данный листинг содержит короткую программу, которая выводит отдельные сведения об использовании памяти и завершается (обычно такой код является частью более крупной программы). Код, выделенный жирным шрифтом, позволяет `dmalloc` сообщать о файлах и номерах строчек с вызовами, приводящими к проблемам.

Листинг 6.11. Выявление ошибки работы с памятью с помощью dmalloc

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

#ifdef DMALLOC
#include "dmalloc.h"
#endif

void usage(char *msg) {
  fprintf(stderr, "%s", msg);
  free(msg);
  return;
}

int main(int argc, char *argv[]) {
  if (argc != 3 && argc != 4) {
    /* сообщение об ошибке будет не длиннее 80 символов */
    char *errmsg = (char *)malloc(80);
    sprintf(
      errmsg,
      "Sorry %s,\nUsage: caesar secret_file keys_file [output_file]\n",
      getenv("USER")
    );
    usage(errmsg);
    free(errmsg);
    exit(EXIT_FAILURE);
  }
  //---snip---

  exit(EXIT_SUCCESS);
}
```

Вывод я покажу чуть позже, а пока необходимо прояснить несколько моментов. В состав `dmalloc` входит утилита командной строки. Вы можете получить подробности о том, как ее использовать, запустив ее следующим образом:

```
% dmalloc --usage
```

Прежде чем отлаживать свою программу с помощью `dmalloc`, введите в командной строке следующее:

```
% dmalloc -l logfile -i 100 low
```

Эта команда указывает `logfile` в качестве имени файла с логом и заставляет библиотеку проводить проверки через каждые 100 вызовов, в соответствии с аргументом `-i`. Если указать для `-i` большее число, то `dmalloc` будет проверять кучу не так часто и ваш код будет работать быстрее; чем меньше данное значение, тем выше вероятность выявления проблем с памятью. Третий аргумент командной строки (в данном случае `low`) определяет количество возможностей отладки; для минимальной проверки следует указать `runtime`, а для более обширного анализа кучи предусмотрены `medium` и `high`.

После того как эта команда будет выполнена, мы можем скомпилировать нашу программу с помощью GCC, как показано ниже:

```
% gcc -DDMALLOC caesar.c -ocaesar -ldmalloc
```

При запуске программы вы должны увидеть следующую ошибку:

```
% ./caesar
Sorry student,
Usage: caesar secret_file keys_file [output_file]
debug-malloc library: dumping program, fatal error
  Error: tried to free previously freed pointer (err 61)
Aborted (core dumped)
```

А в журнальном файле вы найдете такую информацию:

```
% more logfile
1571549757: 3: Dmalloc version '5.5.2' from 'https://dmalloc.com/'
1571549757: 3: flags = 0x4e48503, logfile 'logfile'
1571549757: 3: interval = 100, addr = 0, seen # = 0, limit = 0
1571549757: 3: starting time = 1571549757
1571549757: 3: process pid = 29531
1571549757: 3:   error details: finding address in heap
```

```
1571549757: 3:   pointer '0x7ff010812f88' from 'caesar.c:29' prev access
'unknown'
1571549757: 3: ERROR: free: tried to free previously freed pointer (err 61)
```

Сообщения говорят о том, что мы дважды пытались освободить память, на которую указывает `errmsg`: сначала в функции `usage`, а затем в `main`. Это уязвимость двойного освобождения. Конечно, это лишь один из дефектов, которые может обнаружить `dmalloc`, и в данной демонстрационной программе есть другие ошибки. Другие средства динамического анализа, а также рекомендации по их применению рассматриваются в главе 11.

Системы с повышенными требованиями к безопасности

Во многих системах с повышенными требованиями к безопасности запрещено использовать динамическую память, поскольку менеджеры памяти могут вести себя непредсказуемо и существенно влиять на производительность. Заставляя все приложения работать в фиксированном, заранее выделенном участке памяти, мы можем избавиться от многих из этих проблем и упростить отладку. В отсутствие рекурсии, `alloca` и массивов переменной длины (которые тоже запрещены в таких системах) максимально необходимый размер стека можно вычислить статически; это позволяет доказать, что, независимо от ввода, для выполнения функций приложения достаточно какого-то определенного объема памяти.

GCC также поддерживает флаги `-Wvla` и `-Wvla-larger-than=`*`byte-size`*: первый предупреждает об использовании VLA, а второй сообщает об объявлениях VLA, либо не имеющих ограничений, либо ограниченных аргументом, который допускает размеры массива, превышающие *byte-size* байт.

Упражнения

Попробуйте самостоятельно выполнить следующие упражнения.

1. Исправьте ошибку с использованием освобожденной памяти в листинге 6.4.

2. Проведите дополнительное тестирование программы из листинга 6.11 с помощью `dmalloc`. Попробуйте передавать программе разный ввод, чтобы обнаружить другие дефекты работы с памятью.

Резюме

В этой главе вы научились работать с памятью, имеющей динамический срок хранения, и узнали, чем она отличается от объектов, срок хранения которых определяется автоматически или статически. Мы обсудили кучу, менеджеры памяти и все стандартные функции для работы с ними. Мы определили некоторые распространенные источники ошибок, связанных с использованием динамической памяти, включая утечки и уязвимости двойного освобождения, и предложили несколько способов их предотвращения.

Кроме того, мы затронули более узкие аспекты выделения памяти, такие как структуры с массивами произвольной длины, функция `alloca` и массивы переменной длины. В завершение главы была рассмотрена отладка проблем динамически выделяемой памяти с использованием библиотеки `dmalloc`.

В следующей главе речь пойдет о символах и строках.

7 Символы и строки

Строки — настолько важный и полезный тип данных, что в том или ином виде реализованы почти во всех языках программирования. Поскольку зачастую с их помощью представляют текст, они составляют большую часть данных, которыми пользователь обменивается с программой, включая текстовые поля ввода, аргументы командной строки, переменные окружения и консольный ввод.

В C строковый тип данных основан на концепции формальных строк (Хопкрофт, 1979):

> «Пусть Σ — непустое конечное множество символов, которое называется алфавитом. Тогда строка — конечная последовательность символов из Σ. Например, если Σ = {0, 1}, то 01011 — строка по Σ».

В этой главе мы обсудим различные наборы символов (или *алфавиты* в определении формальной строки), включая ASCII и Unicode, из которых можно составлять строки. Вы познакомитесь с методами представления и изменения строк с помощью функций из стандартной библиотеки C, а также с интерфейсами, проверяющими ограничения, и API POSIX и Windows.

Символы

Цифровые системы, работающие с битами, не имеют естественного понимания символов, которые мы используем в общении. Для обработки таких символов они применяют *кодировки*, в которых для обозначения каждого

отдельного символа предусмотрено целочисленное значение, так называемая *кодовая позиция*. Как вы сами увидите, один и тот же номинальный символ можно закодировать несколькими способами. В реализациях языка C для кодирования символов, как правило, применяют стандарты Unicode, ASCII, Extended ASCII, ISO 8859-1, Shift-JIS и EBCDIC[1].

ASCII

Семибитная версия ASCII (American Standard Code for Information Interchange) описывает 128 символов и их закодированные представления (ANSI X3.4-1986). Символы до `0x1f` включительно, такие как null, backspace и табуляция, являются управляющими. Символы с `0x20` по `0x7e`, включая буквы и цифры, являются печатными.

Этот стандарт зачастую называют *US-ASCII*, чтобы уточнить, что он был разработан в Соединенных Штатах, и основное внимание в нем уделяется типографическим символам, которые используются в данной стране. На US-ASCII основано большинство современных кодировок, хотя они поддерживают множество дополнительных символов.

Символы в диапазоне `0x80-0xFF` не определены в US-ASCII, но входят в восьмибитную кодировку под названием *Extended ASCII*. Символы в этих диапазонах кодируются множеством разных способов в зависимости от кодовой страницы. *Кодовая страница* — это кодировка, которая назначает набору печатных и управляющих символов уникальные номера.

Unicode

Unicode (юникод) — это универсальный стандарт кодирования символов для представления текста в компьютерных программах. Он поддерживает куда более широкий набор символов, чем ASCII; текущий стандарт Unicode (Unicode 2020) охватывает диапазон от `U+0000` до `U+10FFFF`, что выражается в 21-битном кодовом пространстве. Каждое значение в юникоде начинается с префикса `U+`, за которым в печатном тексте идет как минимум четыре шестнадцатеричных цифры. Юникодные символы с `U+0000` по `U+007F` полностью повторяют US-ASCII, а диапазон с `U+0000` по `U+00FF` идентичен кодировке ISO 8859-1 (Latin 1) и содержит символы

[1] Unicode и ASCII упоминаются непосредственно в стандарте C.

из латиницы, которые используются в Северной и Южной Америке, Западной Европе, Океании и большей части Африки.

В Unicode кодовые позиции делятся на *плоскости* — непрерывные группы из 65 536 кодовых позиций. Существует всего 17 плоскостей с номерами от 0 до 16. Символы, использующиеся чаще всего, включая те, которые входят в основные стандарты кодирования, предшествовавшие Unicode, находятся в первой, *основной многоязычной плоскости* (basic multilingual plane, BMP; с `0x0000` по `0xFFFF`), известной также как Plane 0.

Существует также несколько *форматов преобразования Unicode* (Unicode transformation formats, UTF). UTF — это формат кодирования символов, который привязывает каждое скалярное значение Unicode к уникальной последовательности единиц кодирования. *Скалярным значением* может быть любая кодовая позиция Unicode, за исключением верхней и нижней частей суррогатных пар. *Единица кодирования* — это минимальное сочетание битов, способное представить закодированный текст для последующей обработки или обмена. В стандарте Unicode предусмотрено три формата UTF с единицами кодирования разных размеров:

- *UTF-8* — каждый символ представлен последовательностью восьмибитных единиц кодирования в количестве от одной до четырех;
- *UTF-16* — каждый символ представлен последовательностью из одной или двух 16-битных единиц кодирования;
- *UTF-32* — каждый символ представлен одной 32-битной единицей кодирования.

Кодировка UTF-8 преобладает в операционных системах POSIX. Она имеет следующие полезные свойства:

- символы US-ASCII (с `U+0000` по `U+007F`) в ней кодируются одним байтом в диапазоне от `0x00` до `0x7F`. Это значит, что файлы и строки с семибитными символами ASCII будут иметь одну и ту же кодировку как в ASCII, так и в UTF-8;
- использование нулевого байта для завершения строки (эту тему мы обсудим позже) работает так же, как и в строках ASCII;
- все кодовые позиции, определенные в Unicode на сегодняшний день, можно закодировать, используя от одного до четырех байт;

- Unicode позволяет легко определять границы символов в любом направлении путем поиска четко определенных сочетаний битов.

В Windows при компиляции и компоновке программы в Visual C++ можно указать флаг `/utf8`, который переведет исходные и исполняемые символы в UTF-8. Вам также нужно настроить в Windows Unicode UTF-8 для поддержки языков мира (это может измениться в будущем).

В настоящее время основной кодировкой в операционных системах Windows является UTF-16. Как и UTF-8, UTF-16 поддерживает символы разной длины. Как уже упоминалось, BMP состоит из символов в диапазоне от `U+0000` до `U+FFFF`. Символы, кодовые позиции которых больше `U+FFFF`, называются *дополнительными* и определяются парами единиц кодирования под названием *«суррогаты»*. Первая единица кодирования берется из верхнего суррогатного диапазона (`U+D800-U+DBFF`), а вторая — из нижнего (`U+DC00-U+DFFF`).

В отличие от других форматов преобразования Unicode, имеющих переменную длину, UTF-32 — это кодировка фиксированной длины. Ее основное преимущество заключается в том, что кодовые позиции Unicode в ней индексируются напрямую; это значит, что вы можете найти *n*-ю позицию в заданной последовательности за константное время $O(1)$. Для сравнения: в кодировках переменной длины поиск *n*-й кодовой позиции в последовательности подразумевает последовательный перебор.

Исходная и исполняемая кодировки

Во времена, когда язык C впервые проходил стандартизацию, не существовало какой-либо общепринятой кодировки, поэтому он был рассчитан на работу с самыми разнообразными представлениями символов. Вместо того чтобы стандартизировать кодировку, как это делает Java, каждая реализация C определяет два набора символов: *исходные*, с помощью которых написаны исходные файлы, и *исполняемые*, используемые для символов и строковых литералов на этапе компиляции.

Исходная и исполняемая кодировки должны содержать коды для строчных и прописных букв латинского алфавита, 10 десятичных цифр, 29 графических символов, пробел, горизонтальную и вертикальную табуляцию, прогон страницы и перевод строки. В исполняемую кодировку также входят звуковой сигнал, backspace, возврат каретки и null.

Преобразования символов и функции категоризации (такие как `isdigit`) выполняются в момент вызова с учетом региональных настроек. *Региональные настройки* определяют национальные, культурные и языковые особенности.

Типы данных

В языке C предусмотрено несколько типов данных для представления символов, и некоторые из них вам уже встречались. В частности, *узкие символы* в C представлены стандартным типом `char`, а *широкие* — типом `wchar_t` (они могут занимать больше восьми бит).

Тип char

Как мы уже видели, `char` — целочисленный тип, но наличие знака в нем зависит от реализации; то есть в переносимом коде нельзя делать предположения о том, имеет знак этот тип или нет.

Тип `char` следует использовать для символов (в этом случае наличие или отсутствие знака не имеет значения), но не для целочисленных данных (где знак играет важную роль). Данный тип позволяет безопасно представлять семибитные кодировки, такие как US-ASCII. В подобных кодировках старший бит всегда равен 0, поэтому вам не нужно беспокоиться о расширении знака, когда значение типа `char` приводится к `int` и в текущей реализации имеет знак.

Тип `char` также можно использовать для представления восьмибитных кодировок наподобие Extended ASCII, ISO/IEC 8859, EBCDIC и UTF-8, но в реализациях, в которых `char` является восьмибитным типом со знаком, это чревато проблемами. Например, следующий код выводит строку `end of file` при обнаружении конца файла:

```
char c = 'ÿ'; // extended character
if (c == EOF) puts("end of file");
```

Если в качестве исполняемой кодировки используется ISO/IEC 8859-1, то маленькая буква латинского алфавита `y` с диерезисом (ÿ) будет иметь код `255` (`0xFF`). В реализациях, в которых `char` имеет знак, буква `c` будет расширена до `signed int`, что сделает символ ÿ неотличимым от индикатора конца файла, так как они оба будут иметь одно и то же представление.

Похожие проблемы возникают при использовании функций категоризации символов, определенных в <ctype.h>. Эти библиотечные функции принимают в качестве аргумента код символа с типом int или макрос EOF и возвращают true, если значение входит в соответствующий набор символов. Например, функция isdigit проверяет, является ли символ десятичной цифрой с учетом текущих региональных настроек. Любое значение аргумента, которое не является корректным символом или макросом EOF, приводит к неопределенному поведению.

Чтобы избежать неопределенного поведения при вызове этих функций, приведите c к unsigned char перед повышением разрядности, как показано ниже:

```
char c = 'ÿ';
if (isdigit((unsigned char)c)) {
  puts("c is a digit");
}
```

Значение, хранящееся в c, расширяется до signed int, а дополнительные разряды заполняются нулями. Это устраняет возможность неопределенного поведения, поскольку результат по-прежнему можно представить в виде unsigned char.

Тип int

Используйте тип int для данных, которые могут содержать индикатор конца файла (неотрицательное значение) или символы, интерпретируемые как unsigned char и затем приводимые к int. Этот тип возвращается функциями, которые читают символы из потока, включая fgetc, getc, getchar и ungetc. Как мы уже видели, функции из <ctype.h> также принимают аргументы типа int, поскольку в эти функции может быть передан результат вызова fgetc или похожей функции.

Тип wchar_t

Целочисленный тип wchar_t был добавлен в язык C для работы с символами расширенных кодировок. Он может быть со знаком или без и имеет диапазон от WCHAR_MIN включительно до WCHAR_MAX включительно — от реализации зависит как знаковость, так и диапазон. В большинстве случаев wchar_t занимает либо 16, либо 32 бита, но в реализациях, которые

не поддерживают региональные особенности, может иметь ту же ширину, что и `char`. В языке C широкие строки не поддерживают кодировки переменной длины (хотя в Windows на практике используется UTF-16). Реализации могут содержать объявление макроса `__STDC_ISO_10646__`, представляющего целочисленную константу вида *`yyyymmL`* (например, `199712L`); в этом случае тип `wchar_t` представляет символы Unicode, соответствующие заданной версии стандарта. Реализации с 16-битным типом `wchar_t` не удовлетворяют требованиям к определению `__STDC_ISO_10646__` для кодировки ISO/IEC 10646 в версиях Unicode выше 3.1 (ISO/IEC 10646-1:2000 и ISO/IEC 10646-2:2001). Следовательно, для определения `__STDC_ISO_10646__` нужен либо тип `wchar_t`, превышающий 20 бит, либо его 16-битная версия, если `__STDC_ISO_10646__` имеет значение ниже `200103L`. Тип `wchar_t` можно использовать не только для Unicode, но и для других кодировок, включая расширенный код EBCDIC.

Написание переносимого кода с помощью `wchar_t` может быть непростой задачей, поскольку целый ряд аспектов поведения этого типа зависит от реализации. Например, Linux обычно использует 32-битный беззнаковый целочисленный тип, а Windows — 16-битный. Код, вычисляющий длину и размер строк с широкими символами, чреват ошибками и требует к себе дополнительного внимания.

Типы char16_t и char32_t

В более новых языках (таких как Ada95, Java, TCL, Perl, Python и C#) предусмотрены типы данных для юникодных символов. В C11 были добавлены 16- и 32-битные символьные типы `char16_t` и `char32_t` (`<uchar.h>`) для поддержки кодировок UTF-16 и UTF-32 соответственно. C11 не предоставляет библиотечных функций для работы с ними; исключение составляет лишь один набор функций для преобразования символов, которые позволяют разработчикам сторонних библиотек реализовывать все остальные функции. В результате практическая польза от применения этих типов невелика.

В языке C определено два макроса среды, которые позволяют определить, как закодированы символы, представленные этими типами. Если макрос среды `__STDC_UTF_16__` имеет значение `1`, то символы `char16_t` кодируются как UTF-16. Если макрос среды `__STDC_UTF_32__` имеет значение `1`, то символы `char32_t` кодируются как UTF-32. Если макрос не определен, то

используется другая кодировка, определяемая реализацией. В Visual C++ эти макросы не определены.

Символьные константы

Язык C позволяет указывать *символьные константы*, также известные как *символьные литералы* — последовательности из одного символа или более в одинарных кавычках (например, `'ÿ'`). Благодаря символьным константам вы можете указывать значения символов в исходном коде своей программы. В табл. 7.1 показаны виды символьных констант, которые можно указывать в C.

Таблица 7.1. Виды символьных констант

Префикс	Тип
Нет	int
L'a'	Беззнаковый тип, соответствующий wchar_t
u'a'	char16_t
U'a'	char32_t

Самый необычный аспект табл. 7.1 состоит в том, что символьная константа без префикса, такая как `'a'`, имеет тип `int`, а не `char`. В языке C так сложилось, что символьные константы, которые состоят из одного символа или управляющей последовательности, представлены объектом типа `char`, приведенным к `int`. Это отличается от C++, где такие одиночные символьные литералы имеют тип `char`.

Значение символьной константы, содержащей больше одного символа (например, `'ab'`), зависит от реализации. То же самое относится к значению исходного символа, который нельзя представить одной единицей кодирования в исполняемой кодировке. Одним из таких случаев является ранее упомянутый пример с `'ÿ'`. Если исполняемой кодировкой выступает UTF-8, то для представления значения кодовой позиции `U+00FF` может понадобиться две единицы кодирования, поэтому код может выглядеть как `0xC3BF`. C2x добавляет к символьным литералам префикс `u8` для обозначения кодировки UTF-8. Но пока этот стандарт не вышел, у символьных литералов UTF-8 нет префиксов, если только разработчики конкретной реализации не решат добавить их заблаговременно.

Управляющие последовательности

Одинарная кавычка (') и обратная косая черта (\) имеют специальное назначение, поэтому их нельзя представить напрямую в виде символов. Вместо этого одинарная кавычка записывается как \', а обратная косая черта — как \\. С помощью управляющих последовательностей можно представлять другие символы, включая вопросительный знак (?) и произвольные целочисленные значения. Это показано в табл. 7.2.

Таблица 7.2. Управляющие последовательности

Символ	Управляющая последовательность
Одинарная кавычка	\'
Двойная кавычка	\"
Вопросительных знак	\?
Обратная косая черта	\\
Звуковой сигнал	\a
Backspace	\b
Прогон страницы	\f
Перевод строки	\n
Возврат каретки	\r
Горизонтальная табуляция	\t
Вертикальная табуляция	\v
Восьмеричный символ	\<*до трех восьмеричных цифр*>
Шестнадцатеричный символ	\x *шестнадцатеричные цифры*

Управляющими последовательностями, которые состоят из обратной косой черты и строчной буквы, представлены следующие неграфические символы: \a (звуковой сигнал), \b (backspace), \f (прогон страницы), \n (перевод строки), \r (возврат каретки), \t (горизонтальная табуляция) и \v (вертикальная табуляция).

Восьмеричные цифры можно объединять в восьмеричные управляющие последовательности, чтобы получить одиночный символ для символьной константы или одиночный широкий символ для широкосимвольной константы. Численное значение восьмеричного целого определяет код нужного символа (обычного или широкого). *Обратная*

косая черта, за которой следуют цифры, всегда интерпретируется как восьмеричное значение. Например, символ backspace (8 в десятеричной системе) можно представить в виде восьмеричного кода `\10` (или `\010`, что одно и то же).

Точно так же вслед за `\x` можно указать шестнадцатеричные цифры, чтобы получить одиночный символ (обычный или широкий) для символьной константы; его код определяется числовой величиной шестнадцатеричного значения. Например, символ возврата на шаг можно представить в виде шестнадцатеричного кода `\x8` (или `\x08`, что одно и то же).

Linux

В разных операционных системах кодировки символов развивались поразному. До появления UTF-8 в Linux обычно применяли различные расширения для ASCII, в зависимости от языка. Самыми популярными из них были ISO 8859-1 и ISO 8859-2 (в Европе), ISO 8859-7 (в Греции), KOI-8/ISO 8859-5/CP1251 (в России), EUC и Shift-JIS (в Японии), и BIG5 (на Тайване). Создатели дистрибутивов и разработчики приложений постепенно избавляются от этих устаревших кодировок, стараясь представлять локализованные текстовые строки в UTF-8 (Кун, 1999).

GCC поддерживает несколько флагов для настройки кодировок. Вот несколько штук, которые могут вам пригодиться:

```
-fexec-charset=кодировка
```

Флаг `-fexec-charset` устанавливает исполняемую кодировку, с помощью которой интерпретируются строки и символьные константы. По умолчанию используется UTF-8. Вы можете указать любую кодировку, поддерживаемую системной библиотекой `iconv` (более подробно о ней мы поговорим позже в этой главе). Например, флаг `-fexec-charset=IBM1047` заставляет GCC интерпретировать строковые константы, содержащиеся в исходном коде (такие как строки форматирования `printf`) в соответствии с кодовой страницей 1047 кодировки EBCDIC.

Чтобы выбрать исполняемую кодировку для широких строк и символьных констант, используйте флаг `-fwide-exec-charset`:

```
-fwide-exec-charset=кодировка
```

По умолчанию используется UTF-32 или UTF-16, в зависимости от ширины `wchar_t`.

Чтобы установить кодировку ввода, которая используется для перевода содержимого входного файла в исходную кодировку, применяемую в GCC, используйте флаг `-finput-charset`:

```
-finput-charset=кодировка
```

Clang поддерживает флаги `-fexec-charset` и `-finput-charset`, но не `-fwide-exec-charset`, позволяя указать в качестве кодировки только UTF-8. Попытки установить любую другую кодировку отклоняются.

Windows

Поддержка кодировок символов в Windows развивалась беспорядочно. Программы, разработанные для этой ОС, совместимы с кодировками либо на основе Unicode, либо с интерфейсами, которые косвенно используют региональные методы кодирования. В большинстве современных приложений по умолчанию следует выбирать интерфейсы Unicode, чтобы получить предсказуемое поведение во время обработки текста. В целом такой код будет иметь лучшую производительность, поскольку узкие строки, которые передаются библиотечным функциям Windows, зачастую переводятся в Unicode.

Точки входа main и wmain

Visual C++ поддерживает две точки входа в программу: `main`, которая позволяет передавать в качестве аргументов узкие строки, и `wmain`, которая принимает широкосимвольные аргументы. Обе точки имеют похожие форматы объявления формальных параметров, представленные в табл. 7.3.

Таблица 7.3. Объявления точек входа в Windows-программу

Узкосимвольные аргументы	Широкосимвольные аргументы

`int main(void);`
`int main(int argc, char *argv[]);`
`int main(int argc, char *argv[],`
`  char *envp[]);`
	`int wmain(void);`
	`int wmain(int argc, wchar_t *argv[]);`
	`int wmain(int argc, wchar_t *argv[],`
	`  wchar_t *envp[]);`

Какой бы ни была точка входа, выбор кодировки в конечном счете зависит от вызывающего процесса. Однако функции `main` обычно принято передавать дополнительные аргументы и переменные среды в виде указателей на текст, закодированный с использованием текущей кодовой страницы Windows (известной как ANSI), тогда как `wmain` обычно получает текст в кодировке UTF-16.

Если программа запускается из командной оболочки, то интерпретатор последней переводит аргументы в подходящую для этой точки входа кодировку. Процесс Windows начинает работу с командной строки и кодировки UTF-16. Загрузочный код, сгенерированный компилятором/компоновщиком, вызывает функцию `CommandLineToArgvW`, чтобы преобразовать командную строку в формат `argv`, необходимый для вызова `main`, или передает аргументы командной строки напрямую в `argv`, как того требует `wmain`. Затем в вызове `main` результаты приводятся к текущей кодовой странице Windows, которая может зависеть от конкретной системы. Символ `?` из ASCII подставляется вместо символов, не представленных на текущей кодовой странице Windows.

При записи данных в консоль Windows использует кодовую страницу, предусмотренную оригинальным производителем оборудования (original equipment manufacturer, OEM). В разных системах могут применяться разные кодировки, но все они, как правило, отличаются от кодовой страницы Windows. Например, в версии Windows с американским вариантом английского языка кодовой страницей может быть Windows Latin 1, тогда как в качестве кодовой страницы OEM может использоваться DOS Latin US. В целом для записи текстовых данных в `stdout` или `stderr` их сначала необходимо преобразовать в кодовую страницу OEM. В качестве альтернативы можно изменить кодовую страницу консоли, чтобы она совпадала с кодировкой выводимого текста. Если не сделать ни того ни другого, то консольный вывод может оказаться непредсказуемым. Но даже в случае совпадения кодировок программы и консоли символы все равно могут быть выведены неправильно по каким-то другим причинам — например, текущий шрифт, выбранный для консоли, может не содержать подходящих глифов, необходимых для представления символов. Кроме того, по историческим причинам консоль Windows не умеет отображать символы, не входящие в Unicode BMP, поскольку для хранения символа в каждой ячейке выделяется лишь 16-битное значение.

Сравнение узких и широких символов

У каждого системного API в Win32 SDK есть две версии: узкая, с суффиксом `A` (ANSI), и широкая, с суффиксом `W`:

```
int SomeFuncA(LPSTR SomeString);
int SomeFuncW(LPWSTR SomeString);
```

Вы должны определиться с тем, какие символы будет использовать ваше приложение: широкие (UTF-16) или узкие, и затем уже писать соответствующий код. Рекомендуется напрямую вызывать узкую или широкую версию каждой функции и передавать ей строку соответствующего типа:

```
SomeFuncW(L"String");
SomeFuncA("String");
```

В качестве примеров реальных функций из Win32 SDK можно привести `MessageBoxA`/`MessageBoxW` и `CreateWindowExA`/`CreateWindowExW`.

Преобразование символов

Несмотря на то что международный текст все чаще кодируется в Unicode, мы по-прежнему можем встретить кодировки, зависящие от языка или страны, поэтому их необходимо как-то преобразовывать. В частности, Windows все еще работает с традиционными кодировками, такими как IBM EBCDIC и ISO 8859-1, которые имеют ограниченные наборы символов. При выполнении ввода/вывода программам часто приходится переводить текст из Unicode в традиционные кодировки и обратно.

Не все символы можно закодировать методами, которые зависят от языка или страны. Это довольно очевидно в случае с кодировкой US-ASCII, которая неспособна представить символы, занимающие больше семи бит. Latin 1 никогда не сможет как следует закодировать символ 晥, а многие неяпонские буквы и слова невозможно преобразовать в Shift-JIS без потери информации.

В следующих подразделах описаны различные подходы к переводу символов из одной кодировки в другую.

Стандартная библиотека C

Стандартная библиотека C предоставляет ряд функций для выполнения преобразований между узкими и широкими единицами кодирования (`char`

и wchar_t). Функции mbtowc (многобайтный символ в широкий), wctomb (широкий символ в многобайтный), mbrtowc (многобайтный символ в широкий (перезапускаемая)) и wcrtomb (широкий символ в многобайтный (перезапускаемая)) преобразуют по одной единице кодирования за раз, записывая результат в выходной объект или буфер. Функции mbstowcs (многобайтная строка в широкую), wcstombs (широкая строка в многобайтную), mbsrtowcs (многобайтная строка в широкую (перезапускаемая)) и wcsrtombs (широкая строка в многобайтную (перезапускаемая)) выполняют посимвольное преобразование строк, записывая результат в выходной буфер.

Чтобы выполнять последовательные преобразования должным образом, промежуточные результаты вызовов этих функций нужно как-то сохранять. *Неперезапускаемые (nonrestartable)* версии хранят свое состояние внутри, а у *перезапускаемых (restartable)* есть дополнительный параметр, представляющий собой указатель на объект типа mbstate_t, который описывает текущее состояние преобразования соответствующей последовательности многобайтных символов. Данный объект хранит данные о состоянии, что позволяет возобновить преобразование с того места, где оно было остановлено ради вызова другой функции, относящейся к какой-то другой задаче. *Строковые* версии предназначены для выполнения пакетных преобразований сразу нескольких единиц кодирования.

У этих функций есть несколько ограничений. Как уже упоминалось ранее, в Windows для wchar_t используются 16-битные единицы кодирования. Это может быть проблемой, поскольку стандарт C требует, чтобы объект wchar_t умел представить любой символ в рамках текущих региональных настроек, и 16 бит для этого может не хватить. Формально язык C не позволяет задействовать несколько объектов типа wchar_t для представления одного символа. Следовательно, применение стандартных функций преобразования может приводить к потере данных. С другой стороны, в большинстве реализаций POSIX для wchar_t применяются 32-битные единицы кодирования, что позволяет использовать UTF-32. Поскольку одна единица кодирования UTF-32 способна представить целую кодовую позицию, преобразование с помощью стандартных функций не может вызвать потерю или усечение данных.

В ответ на возможную потерю данных при использовании стандартных функций преобразования комитет во главе стандарта C добавил в C11 следующие вызовы:

- mbrtoc16, c16rtomb — выполняют преобразования между узкими единицами кодирования и одной или несколькими единицами char16_t;

- `mbrtoc32, c32rtomb` — преобразуют последовательность узких единиц кодирования в одну или несколько единиц `char32_t`.

Первые две функции выполняют преобразования между региональными кодировками, представленными в виде массивов `char`, и данными UTF-16, хранящимися в массиве `char16_t` (при условии, что `__STDC_UTF_16__` равно `1`). Вторая пара функций выполняет преобразования между региональными кодировками и данными UTF-32, хранящимися в массиве `char32_t` (при условии, что `__STDC_UTF_32__` равно `1`). Программа, показанная в листинге 7.1, использует функцию `mbrtoc16`, чтобы преобразовать входную строку UTF-8 в текст, закодированный с помощью UTF-16.

Листинг 7.1. Использование функции mbrtoc16 для преобразования строки из UTF-8 в char16_t

```
#include <locale.h>
#include <uchar.h>
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

❶ #if __STDC_UTF_16__ != 1
  #error "__STDC_UTF_16__ not defined"
  #endif

  int main(void) {
  ❷ setlocale(LC_ALL, "en_US.utf8");
    char input[] = u8"I ♥ 🌽s!";
    const size_t input_size = sizeof(input);
    char16_t output[input_size]; // UTF-16 требует меньше единиц
                                 // кодирования, чем UTF-8
    char *p_input = input;
    char *p_end = input + input_size;
    char16_t *p_output = output;
    size_t code;
    mbstate_t state = {0};
    puts(input);
  ❸ while ((code = mbrtoc16(p_output, p_input, p_end-p_input, &state))){
      if (code == (size_t)-1)
        break; // -1 – обнаружена некорректная последовательность
               // единиц кодирования
      else if (code == (size_t)-2)
        break; // -2 – пропущенные элементы в последовательности
               // единиц кодирования
      else if (code == (size_t)-3)
        p_output++; // -3 – верхняя часть суррогатной пары
      else {
```

```
      p_output++; // одно значение было записано
      p_input += code; // code — это число единиц кодирования,
                       // прочитанных функцией
    }
  }
  size_t output_size = p_output - output + 1;
  printf("Converted to %zu UTF-16 code units: [ ", output_size);
  for(size_t x = 0; x < output_size; ++x) printf("%#x ", output[x]);
  puts("]");
}
```

Мы вызываем функцию `setlocale` ❷, чтобы указать UTF-8 в качестве многобайтной кодировки, передавая ей строку, определенную текущей реализацией. Директивы препроцессора ❶ следят за тем, чтобы макрос `__STDC_UTF_16__` имел значение `1` (более подробно о директивах препроцессора поговорим в главе 9). В итоге каждый вызов функции `mbrtoc16` переводит отдельную кодовую позицию из UTF-8 в UTF-16. Если полученная единица кодирования UTF-16 принадлежит к верхней части суррогатной пары, то объект `state` обновляется, сигнализируя о том, что следующий вызов `mbrtoc16` запишет нижнюю часть суррогатной пары без учета входной строки.

У `mbrtoc16` нет строковой версии, поэтому мы циклически перебираем входную строку UTF-8 и вызываем `mbrtoc16` ❸, чтобы перевести ее в UTF-16. Если в ходе кодирования произойдет ошибка или в последовательности единиц кодирования обнаружатся недостающие элементы, функция `mbrtoc16` вернет `(size_t)-1` или соответственно `(size_t)-2`. В обоих случаях цикл прерывается и преобразование заканчивается.

Возвращение `(size_t)-3` означает, что функция вывела верхнюю часть суррогатной пары и затем сохранила индикатор в параметре `state`. Этот индикатор будет использован при следующем вызове функции `mbrtoc16`, чтобы она могла вернуть нижнюю часть суррогатной пары и завершить формирование последовательности `char16_t`, представляющей отдельную кодовую позицию. Все перезапускаемые функции преобразования в стандартной библиотеке C ведут себя похожим образом по отношению к параметру `state`.

Если функция вернет что-то кроме `(size_t)-1`, `(size_t)-2` или `(size_t)-3`, то указатель `p_output` инкрементируется, а `p_input` увеличивается на количество единиц кодирования, прочитанных функцией и преобразование строки продолжается.

Библиотека libiconv

GNU `libiconv` — это популярная кросс-платформенная, открытая библиотека для перевода строк из одной кодировки в другую. В ее состав входит функция `iconv_open`, выделяющая дескриптор преобразования, с помощью которого вы можете менять кодировки байтовых последовательностей. В документации к данной функции (см. на https://www.gnu.org/) перечислены строки для обозначения той или иной региональной кодировки, такие как `ASCII`, `ISO-8859-1`, `SHIFT_JIS` или `UTF-8`.

API Win32 SDK для преобразования строк

Win32 SDK предоставляет две функции для выполнения преобразований между широкосимвольными и узкосимвольными строками:

- `MultiByteToWideChar` — переводит последовательность символов в новую (широкосимвольную) строку UTF-16;
- `WideCharToMultiByte` — переводит (широкосимвольную) строку UTF-16 в новую последовательность символов.

Функция `MultiByteToWideChar` переводит текстовые данные, закодированные с помощью любой кодовой страницы, в строку UTF-16. Функция `WideCharToMultiByte` аналогичным образом переводит текстовые данные, закодированные как UTF-16, в кодировку с любой кодовой страницей. Поскольку не все кодовые страницы могут представить данные в UTF-16, эта функция позволяет указать, какой символ будет по умолчанию подставляться вместо любых символов UTF-16, которые не удается преобразовать.

Строки

В языке C нет поддержки примитивного строкового типа, и она, вероятно, никогда не появится. Вместо этого строки реализованы в виде массивов символов. Они бывают двух видов: узкие и широкие.

Узкая строка имеет тип массива `char`. Она состоит из непрерывной последовательности символов с нуль-символом конца строки. Указатель на строку ссылается на ее первый символ. Размер строки определяется количеством байтов, выделенных для собственно массива символов.

Длина строки — это количество единиц кодирования (байтов) до первого нуль-символа. На рис. 7.1 размер строки равен 7, а длина — 5. К элементам массива, находящимся за нуль-символом, нельзя обращаться. Элементы массива, которые не были инициализированы, нельзя читать.

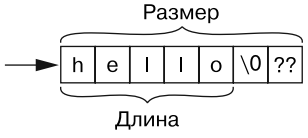

Рис. 7.1. Простая узкая строка

Широкие строки имеют тип массива `wchar_t`. Это непрерывная последовательность широких символов, которая содержит широкий нуль-символ конца строки. *Указатель* на широкую строку ссылается на ее первый широкий символ. *Длина* широкой строки определяется количеством единиц кодирования, находящихся перед первым широким нуль-символом.

На рис. 7.2 показано два представления строки *hello* с порядком следования байтов от старшего к младшему (UTF-16BE) и от младшего к старшему (UTF-16LE). Размер массива определяется реализацией. Данный массив занимает 14 байт, и мы предполагаем, что текущая реализация имеет восьмибитные байты и 16-битный тип `wchar_t`. Длина этой строки равна 5, так как количество символов не изменилось. К элементам массива, находящимся за нуль-символом, нельзя обращаться. Элементы массива, которые не были инициализированы, нельзя читать.

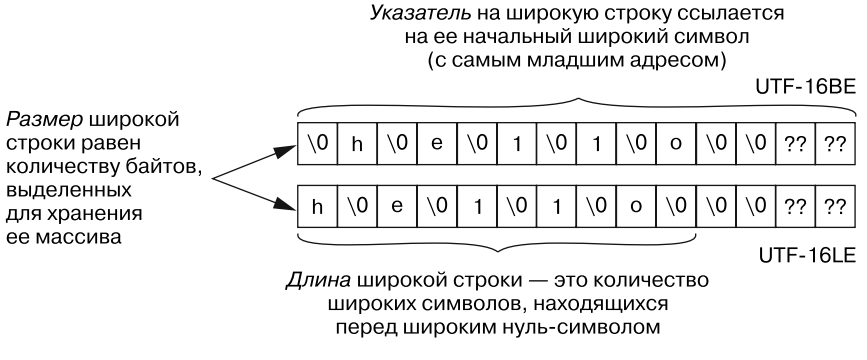

Рис. 7.2. Пример строки в форматах UTF-16BE и UTF-16LE

Строковые литералы

Символьный строковый литерал — это строковая константа, представленная последовательностью произвольного количества многобайтных символов (начиная с нуля), заключенных в двойные кавычки, например `"ABC"`. Для обозначения разных типов символов при объявлении строковых литералов можно указывать специальные префиксы:

- тип строковых литералов `char` — например, `"ABC"`;
- тип строковых литералов `wchar_t` с префиксом `L` — например, `L"ABC"`;
- тип строковых литералов UTF-8 с префиксом `u8` — например, `u8"ABC"`;
- тип строковых литералов `char16_t` с префиксом `u` — например, `u"ABC"`;
- тип строковых литералов `char32_t` с префиксом `U` — например, `U"ABC"`.

Стандарт C не требует от реализаций использования ASCII для строковых литералов. Но вы можете указать префикс `u8`, чтобы строковый литерал кодировался как UTF-8, и если все символы этого литерала входят в ASCII, компилятор сгенерирует строку в кодировке ASCII, даже если реализация обычно использует другую кодировку (такую как EBCDIC).

Строковый литерал имеет неконстантный тип массива[1]. Его изменение считается неопределенным поведением и запрещено правилом STR30-C стандарта CERT C (не пытайтесь изменять строковые литералы). Это объясняется тем, что такие литералы могут храниться в памяти, доступной только для чтения, или же некоторые из них могут разделять одну и ту же память, в результате чего модификация одной строки приводит к изменению других.

Строковыми литералами зачастую инициализируют массивы, количество элементов в которых может быть указано явно и совпадать с количеством символов в литерале. Взгляните на следующее объявление:

```
#define S_INIT "abc"
// ---snip---
const char s[4] = S_INIT;
```

Размер массива `s`, 4, в точности подходит для инициализации с помощью строкового литерала, включая замыкающий нулевой байт.

---

[1] Это исторически сложившийся парадокс. По факту обращаться со строковыми литералами нужно как с константными массивами, а иначе см. далее. В C++ это исправлено, и строковые литералы имеют формальный константный тип. — *Примеч. науч. ред.*

Но если в строковый литерал, который использовался для инициализации массива, добавить еще один символ, то этот код приобретает совершенно иной смысл:

```
#define S_INIT "abcd"
// ---snip---
const char s[4] = S_INIT;
```

Размер массива `s` по-прежнему равен 4, а размер строкового литерала теперь равен 5. В итоге массиву `s` присваивается последовательность символов `"abcd"` без замыкающего нулевого байта. Данный синтаксис предусмотрен специально для того, чтобы вы могли инициализировать символьные массивы, а не строки. Поэтому ваш компилятор вряд ли посчитает это объявление некорректным. Когда такой массив передается в качестве аргумента строковой функции, GCC выдает предупреждение.

Существует определенный риск того, что при обслуживании кода строковый литерал могут случайно заменить символьным массивом без замыкающего нуль-символа, особенно при определении этого литерала отдельно от объявления массива, как в данном примере. Если вы хотите всегда инициализировать `s` с помощью строки, то границы массива указывать не нужно. В таком случае компилятор выделит место, которого хватит для всего строкового литерала, включая замыкающий нуль-символ:

```
const char s[] = S_INIT;
```

Этот подход упрощает обслуживание кода, поскольку размер массива можно определить всегда, даже при изменении размера строкового литерала.

Размер массива, объявленного с помощью данного синтаксиса, можно узнать на этапе компиляции, используя операцию `sizeof`:

```
size_t size = sizeof s;
```

Если бы мы определили эту строку следующим образом:

```
const char *foo = S_INIT;
```

то нам пришлось бы вызывать функцию `strlen`, чтобы получить длину, как показано ниже:

```
size_t size = strlen(foo) + 1U;
```

Это могло бы потребовать дополнительных ресурсов на этапе выполнения.

Функции для работы со строками

Работать со строками в языке C можно несколькими способами, первый из которых состоит в использовании функций из стандартной библиотеки. Функции для работы с узкими и широкими строками определены в заголовочных файлах `<string.h>` и `<wchar.h>` соответственно. В последние несколько лет эти устаревшие функции были связаны с различными уязвимостями безопасности. Дело в том, что они не проверяют размер массива (зачастую им попросту не хватает информации для выполнения таких проверок), поэтому вы сами должны указать размер, достаточный для хранения вывода. Хотя с помощью этих функций можно писать безопасный, надежный и корректный код, они поощряют стиль программирования, который может приводить к переполнениям буферов, если результат оказывается слишком большим для предоставленного массива. Они неопасны сами по себе, но их часто неправильно используют, и при работе с ними нужно проявлять осторожность (или не работать с ними вообще).

В связи с этим в спецификации C11 появилось стандартное (но необязательное) приложение K для интерфейсов с проверкой границ. Это дополнение предоставляет альтернативные библиотечные функции, позволяющие писать более безопасный код; чтобы этого достичь, они, к примеру, требуют указывать длину выходного буфера и проверяют, достаточно ли ее для того, чтобы вместить их вывод. Например, в приложении K определены функции `strcpy_s`, `strcat_s`, `strncpy_s` и `strncat_s`, которые являются заменой вызовов `strcpy`, `strcat`, `strncpy` и `strncat` из стандартной библиотеки C.

POSIX тоже определяет несколько функций для работы со строками, например, `strdup` и `strndup`. Они представляют собой еще один набор строковых API, доступных на POSIX-совместимых платформах, таких как Linux и Unix (IEEE Std 1003.1:2018).

Visual C++ предоставляет все функции для работы со строками, входящие в стандартную библиотеку C вплоть до C99, но не поддерживает полноценную спецификацию POSIX[1]. Многие библиотечные функции в Visual C++ начинаются с подчеркивания, например `_strdup` вместо `strdup`. Visual C++ также поддерживает многие безопасные функции для работы со строками из приложения K и предупреждает об использовании

[1] Более подробную информацию о поддержке POSIX в продуктах Microsoft можно найти в разделе «Совместимость» на странице https://docs.microsoft.com/ru-ru/cpp/c-runtime-library/compatibility?view=vs-2019/.

их опасных альтернатив, если только перед подключением заголовочного файла, в котором они объявлены, не определить `_CRT_SECURE_NO_WARNINGS`.

Каждая из этих библиотек для работы со строками будет рассмотрена в следующих подразделах.

Заголовочные файлы <string.h> и <wchar.h>

В стандартной библиотеке C есть такие общеизвестные функции, как `strcpy`, `strncpy`, `strcat`, `strncat`, `strlen` и т. д., а также вызовы `memcpy` и `memmove`, с помощью которых вы можете копировать и соответственно перемещать строки. В стандарте C также предусмотрен интерфейс для широких символов, рассчитанный на работу с объектами типа `wchar_t` вместо `char` (имена этих функций похожи на имена их узкосимвольных аналогов, только с `wc` вместо `str`, а вызовы для работы с памятью имеют префикс `w`). Примеры функций для работы с узкосимвольными и широкосимвольными строками перечислены в табл. 7.4. Их не следует путать.

Таблица 7.4. Строковые функции для работы с узкими и широкими символами

Узкие (char)	Широкие (wchar_t)	Описание
strcpy	wcscpy	Копия строки
strncpy	wcsncpy	Усеченная (truncated), заполненная нулями копия
memcpy	wmemcpy	Копирует заданное количество единиц кодирования
memmove	wmemmove	Копирует заданное количество единиц кодирования (которые могут пересекаться)
strcat	wcscat	Объединяет строки
strncat	wcsncat	Объединяет строки с усечением
strcmp	wcscmp	Сравнивает строки
strncmp	wcsncmp	Сравнивает определенную часть строк
strchr	wcschr	Ищет символ в строке
strcspn	wcscspn	Возвращает количество символов в начале первой строки, среди которых нет ни одного символа, входящего во вторую строку
strpbrk	wcspbrk	Ищет первое вхождение набора символов в строке
strrchr	wcsrchr	Ищет первое вхождение символа в строке

Продолжение ⇩

Таблица 7.4 *(продолжение)*

Узкие (char)	Широкие (wchar_t)	Описание
strspn	wcsspn	Вычисляет длину сегмента первой строки, состоящего только из символов во второй строке
strstr	wcsstr	Ищет подстроку
strtok	wcstok	Разбивает строку на лексемы (изменяет исходную)
memchr	wmemchr	Ищет в памяти единицу кодирования
strlen	wcslen	Вычисляет длину строки
memset	wmemset	Заполняет память заданными единицами кодирования

Эти функции для работы со строками считаются эффективными, поскольку делегируют управление памятью вызывающей стороне и их можно использовать как со статически, так и с динамически выделенными блоками. Далее мы более подробно обсудим те из функций, которые применяются чаще всего.

Размер и длина

Ранее в данной главе уже упоминалось, что строки имеют как *размер* (количество байтов, выделенных для собственно массива символов), так и *длину*. Размер статически выделенного массива можно определить на этапе компиляции с помощью операции `sizeof`:

```
char str[100] = "Here comes the sun";
size_t str_size = sizeof(str); // str_size равно 100
```

Длину строки можно вычислить с использованием функции `strlen`:

```
char str[100] = "Here comes the sun";
size_t str_len = strlen(str); // str_len равно 18
```

Функция `wcslen` вычисляет длину широкой строки, а именно количество единиц кодирования, находящихся перед нуль-символом конца строки:

```
wchar_t str[100] = L"Here comes the sun";
size_t str_len = wcslen(str); // str_len равно 18
```

Длина — это количество чего-то, но не всегда ясно, чего именно. Ниже представлены отдельные элементы, которые *могли бы* подсчитываться при определении длины строки.

- *Байты* — полезно при выделении памяти.
- *Единицы кодирования* — количество отдельных единиц кодирования, которые представляют строку. Этот показатель зависит от кодировки и тоже может использоваться для выделения памяти.
- *Кодовые позиции* (символы) могут занимать несколько единиц кодирования. При выделении памяти данный показатель бесполезен.
- *Расширенный кластер графем* — группа из одного или более юникодных скалярных значений, приблизительно соответствующая одному символу в том виде, в котором его воспринимают пользователи. Многие отдельные символы, такие как é, 김 и 🇮🇳, могут состоять из нескольких юникодных скалярных значений. Алгоритм определения границ Unicode объединяет эти кодовые позиции в расширенные кластеры графем.

Функции `strlen` и `wcslen` считают единицы кодирования. В случае с `strlen` это соответствует количеству байтов. Определение необходимого места с помощью `wcslen` выглядит не настолько просто, поскольку размер типа `wchar_t` зависит от реализации. В листинге 7.2 показаны примеры динамического выделения памяти для узких и широких строк.

Листинг 7.2. Динамическое выделение памяти для узкосимвольных и широкосимвольных функций

```
// узкие строки
char str1[] = "Here comes the sun";
char *str2 = malloc(strlen(str1) + 1));
// широкие строки
wchar_t wstr1[] = L"Here comes the sun";
wchar_t *wstr2 = malloc((wcslen(wstr1) + 1) * sizeof(wchar_t));
```

Чтобы определить размер узкой строки, можно воспользоваться функцией `strlen` и затем, прежде чем выделять память, прибавить `1` для нуль-терминатора. Если речь идет о широкой строке, то можно вызвать функцию `wcslen`, прибавить `1`, чтобы учесть нуль-терминатор, и затем умножить результат этой операции на размер типа `wchar_t`.

Подсчет кодовых позиций или расширенных кластеров графем нельзя применять при выделении памяти, поскольку результат может состоять из непредсказуемого количества единиц кодирования[1]. С помощью рас-

[1] По адресу https://hsivonen.fi/string-length/ находится интересная презентация о длине строк под названием It's Not Wrong that "🤦".length == 7.

ширенных кластеров графем можно определить, в каком месте обрезать строку, если, к примеру, ей не хватает места. Обрезание по границе расширенного кластера графем позволяет избежать усечения символов в том виде, в котором их воспринимают пользователи.

Вызов функции `strlen` может быть ресурсоемкой операцией, так как ей нужно перебрать элементы массива в поисках нуль-символа. Ниже показана простая реализация этой функции:

```
size_t strlen(const char * str) {
  const char *s;
  for (s = str; *s; ++s) {}
  return s - str;
}
```

Функция `strlen` не может узнать размер объекта, на который указывает `str`. Если вы передадите ей некорректную строку, в пределах которой нет нуль-символа, то она выйдет за границы массива, что приведет к неопределенному поведению. Неопределенное поведение также возникает, когда `strlen` передают нулевой указатель (из-за его разыменовывания). К тому же поведение данной реализации функции `strlen` не определено для строк больше `PTRDIFF_MAX`. Если избегать создания таких объектов, то данная реализация ведет себя нормально.

Функция strcpy

С вычислением размера динамически выделяемой памяти все сложнее. Одно из решений состоит в сохранении размера в момент выделения, чтобы его можно было задействовать позже. В листинге 7.3 используется функция `strcpy`, которая копирует `str` путем определения длины строки и затем прибавляет `1`, чтобы учесть нуль-символ конца строки.

Листинг 7.3. Копирование строки

```
  char str[100] = "Here comes the sun";
  size_t str_size = strlen(str) + 1;
  char *dest = (char *)malloc(str_size);
  if (dest) {
    strcpy(dest, str);
  }
  else {
    /* обрабатываем ошибку */
  }
```

После этого мы можем использовать `str_size`, чтобы динамически выделить память для копии. Функция `strcpy` копирует исходную строку (`str`) в выделенный участок памяти (`dest`), включая нуль-символ конца строки. Функция `strcpy` возвращает адрес начала скопированной строки, который в данном примере игнорируется.

Типичная реализация функции `strcpy` выглядит так:

```
char *strcpy(char *dest, const char *src) {
  char *save = dest;
  while (*dest++ = *src++);
  return save;
}
```

Данный код сохраняет указатель на скопированную строку в переменную `save` (чтобы использовать ее в качестве возвращаемого значения) и затем копирует все байты исходного массива в конечный. Цикл `while` прерывается при копировании первого нулевого байта. Поскольку функции `strcpy` неизвестны ни длина исходной строки, ни размер конечного массива, она исходит из того, что все ее аргументы были проверены вызывающей стороной; это позволяет ей просто копировать каждый байт исходной строки в конечный массив без каких-либо проверок.

Проверка аргументов

Проверку аргументов может выполнять как вызывающая, так и вызываемая функция. Выполнение проверки и там и там считается избыточным и в целом нежелательным стилем защитного программирования. Проверку обычно принято требовать только по одну сторону интерфейса.

Самый эффективный подход с точки зрения скорости — делегировать проверку вызывающему коду, поскольку у него должно быть лучшее представление о состоянии программы. Как можно видеть в листинге 7.3 (см. выше), аргументы `strcpy` не нужно проверять дополнительно: переменная `str` указывает на статически выделенный массив, который был должным образом инициализирован во время объявления, а параметр `dest` представляет собой ненулевой указатель на динамически выделенный участок памяти, размера которого достаточно для хранения копии `str`, включая нуль-символ. Следовательно, вызов `strcpy` безопасен и копирование можно выполнить максимально быстро. Этот подход к проверке аргументов повсеместно используется функциями стандартной

библиотеки, поскольку он соответствует «духу языка C», демонстрируя оптимальную эффективность и доверяя способности программиста передать корректные аргументы.

Более безопасный и экономный с точки зрения занимаемого места метод состоит в том, чтобы аргументы проверял вызываемый код. Таким образом снижается вероятность возникновения ошибок, поскольку проверку выполняет тот, кто реализует библиотечную функцию, поэтому нам больше не нужно надеяться на то, что программист передаст корректные аргументы. Автор функции обычно лучше понимает, какие аргументы необходимо проверять. Если код проверки ввода окажется дефектным, то исправления нужно будет внести лишь в одном месте. Данный подход обычно позволяет сэкономить место. Но, поскольку эти проверки проводятся даже когда не нужны, это может отрицательно сказаться на скорости работы. Зачастую тот, кто делает системные вызовы, проверяет их ввод, даже если они сами потом выполняют аналогичные проверки. Этот метод также предусматривает дополнительную обработку ошибок для функций, которые в настоящий момент не сигнализируют о сбоях, но, по-видимому, должны это делать, если внутри них происходит проверка аргументов. Когда функции передают строковые аргументы, она не всегда может определить, являются ли они корректными строками с нуль-символом конца строки и достаточно ли для копирования места, на которое они указывают.

Из этого можно сделать такой вывод: не стоит предполагать, что функции стандартной библиотеки C сами проверяют свои аргументы, если об этом явно не сказано в стандарте.

Функция memcpy

Функция `memcpy` копирует заданное количество символов, `size`, из объекта, на который указывает `str`, в объект, на который указывает `dest`:

```
void *memcpy(void * restrict dest, const void * restrict src, size_t size);
```

Для копирования строк вместо `strcpy` можно использовать функцию `memcpy`, если размер конечного массива больше или равен аргументу `size`, исходный массив содержит в своих пределах нуль-символ, а длина строки меньше `size - 1` (чтобы итоговая строка содержала в конце нуль-символ). Функцию `strcpy` лучше всего использовать для копирования строк,

а `memcpy` — для копирования нетипизированных участков памяти. Кроме того, помните, что во многих случаях для эффективного копирования объектов подходит операция присваивания (`=`).

Большинство функций в стандартной библиотеке C возвращают указатель на начало строки, переданной в качестве аргумента, чтобы вы могли объединять вызовы таких строковых функций в цепочки. Например, представленная ниже последовательность вложенных вызовов формирует полное имя человека, копируя и затем объединяя его составляющие:

```
strcat(strcat(strcat(strcat(strcpy(full, first), " "), middle), " "), last);
```

Но при объединении подстрок массива `full` его перебор приходится повторять без всякой необходимости; было бы куда более практично, если бы функции возвращали указатели на *конец* измененной строки, чтобы не перебирать ее лишний раз. В спецификации C2x появится функция для копирования строк с более продуманным интерфейсом, `memccpy`. Она уже должна быть доступна в окружениях POSIX, но вам нужно включить ее объявление следующим образом:

```
#define _XOPEN_SOURCE 700
#include <string.h>
```

Функция gets

Функция `gets` принимает ввод, но не позволяет указать размер итогового массива, что является ее недостатком. Из-за этого она не может предотвратить переполнение буфера. По этой причине она была признана устаревшей в C99 и не вошла в C11. Но ввиду ее многолетнего использования большинство библиотек по-прежнему предоставляют ее реализацию для обратной совместимости, и потому она может встречаться в реальном коде. Вы *всегда* должны ее избегать; заменяйте ее в любых проектах, которые вы сопровождаете.

Посмотрим, что делает функцию `gets` настолько неудачной. Код, представленный в листинге 7.4, предлагает пользователю ввести `y` или `n` в зависимости от того, хочет ли он продолжить работу. Если ввести больше восьми символов, то поведение этой программы будет неопределенным. Но так как функция `gets` не может проверить размер итогового массива, она просто запишет данные за пределы его объекта.

Листинг 7.4. Неправильное использование устаревшей функции gets

```
#include <stdio.h>
#include <stdlib.h>
void get_y_or_n(void) {
  char response[8];
  puts("Continue? [y] n: ");
  gets(response);
  if (response[0] == 'n')
    exit(0);
  return;
}
```

В листинге 7.5 показана упрощенная реализация функции `gets`. Как видите, у того, кто ее вызывает, нет возможности ограничить количество считываемых символов.

Листинг 7.5. Реализация функции gets

```
char *gets(char *dest) {
  int c;
  char *p = dest;
  while ((c = getchar()) != EOF && c != '\n') {
    *p++ = c;
  }
  *p = '\0';
  return dest;
}
```

Функция `gets` циклически считывает по одному символу. Цикл прерывается при чтении либо `EOF`, либо символа перевода строки `'\n'`. Если ни того ни другого не обнаружится, то она будет продолжать записывать в массив `dest`, не заботясь о выходе за границы.

В листинге 7.6 показана функция `get_y_or_n` из листинга 7.4, но с вложенным вызовом `gets`.

Листинг 7.6. Плохо написанный цикл while

```
#include <stdio.h>
#include <stdlib.h>
void get_y_or_n(void) {
  char response[8];
  puts("Continue? [y] n: ");
  int c;
  char *p = response;
❶ while ((c = getchar()) != EOF && c != '\n') {
    *p++ = c;
  }
```

```
  *p = '\0';
  if (response[0] == 'n')
    exit(0);
}
```

Теперь циклу `while` ❶ доступен размер итогового массива, однако он не использует эту информацию. При чтении или записи массива подобным образом следует сделать так, чтобы достижение размера итогового массива было условием завершения цикла.

Интерфейсы с проверкой ограничений из приложения K

В приложении K к стандарту C11 появились интерфейсы с проверкой ограничений. Входящие в их состав функции проверяют, является ли выходной буфер достаточно большим для предполагаемого результата, и если нет, то сигнализируют об ошибке. Они предназначены для предотвращения записи данных за пределы массива и добавления нуль-символа в конец каждой итоговой строки. Управление памятью в этих функциях возлагается на вызывающую сторону, и потому выделение места может происходить статически или динамически до их вызова.

Компания Microsoft создала функции C11 приложения K, чтобы упростить модернизацию своей устаревшей кодовой базы в ответ на многочисленные нарушения безопасности, которые получили широкую огласку в 1990-х годах. Затем эти функции были представлены на рассмотрение комитета по стандартизации языка C в виде предложения ISO/IEC TR 24731-1 (ISO/IEC TR 24731-1:2007) и позже вошли в состав C11 в качестве дополнительных расширений. Несмотря на их повышенное удобство использования и безопасность, на момент написания этой книги они реализованы далеко не везде.

Функция gets_s

В интерфейс с проверкой ограничений приложения K входит функция `gets_s`, с помощью которой можно устранить неопределенное поведение в листинге 7.4, спровоцированное использованием `gets`. Листинг 7.7 имеет похожий код, только функция `gets_s` в нем проверяет границы массива. По умолчанию, когда ввод достигает максимального количества символов, реализация сама определяет, что делать дальше, но обычно в этом случае

вызывается функция abort. Вы можете изменить данное поведение с помощью функции set_constraint_handler_s, о которой речь пойдет в пункте «Ограничения времени выполнения» на с. 199.

Листинг 7.7. Использование функции gets_s

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
#include <stdlib.h>

void get_y_or_n(void) {
  char response[8];
  size_t len = sizeof(response);
  puts("Continue? [y] n: ");
  gets_s(response, len);
  if (response[0] == 'n') exit(0);
}
```

В первой строчке листинга 7.7 находится определение макроса __STDC_WANT_LIB_EXT1__, разворачиваемое в значение 1. Затем подключается заголовочный файл, в котором определены интерфейсы с проверкой ограничений; это делает их доступными для использования в нашей программе. В отличие от gets функция gets_s принимает размер в качестве второго аргумента. Таким образом, этот обновленный код вычисляет размер итогового массива с помощью операции sizeof и передает полученное значение функции gets_s. Результатом нарушения ограничений будет поведение, определяемое реализацией.

Функция strcpy_s

Данная функция — близкий аналог функции strcpy из <string.h>. Функция strcpy_s копирует символы из исходной строки в итоговый символьный массив вплоть до (и включительно) нуль-символа конца строки. Вот как выглядит ее сигнатура:

```
errno_t strcpy_s(
  char * restrict s1, rsize_t s1max, const char * restrict s2
);
```

У функции strcpy_s есть дополнительный аргумент типа rsize_t, который определяет максимальную длину конечного буфера. Вызов завершается успешно, только если исходную строку удается полностью скопировать по месту назначения и не переполнить при этом конечный буфер. Функция

strcpy_s следит за тем, чтобы не нарушались следующие ограничения времени выполнения:

- s1 и s2 не должны быть нулевыми указателями;
- s1max не должно быть больше RSIZE_MAX;
- s1max не должно быть равно нулю;
- s1max должно быть больше strnlen_s(s2, s1max);
- копируемые объекты не должны пересекаться.

Чтобы скопировать строку за один проход, типичная реализация функции strcpy_s извлекает каждый символ из исходной строки и помещает его в конечный массив, пока не будут скопированы все символы или пока не заполнится конечный массив. Если строка не помещается целиком, а s1max имеет положительное значение, то функция strcpy_s записывает в первый байт конечного массива нуль-символ, создавая тем самым пустую строку.

Ограничения времени выполнения

Если функция имеет *ограничения времени выполнения*, то обнаруживает их нарушение и вызывает соответствующий обработчик. Если он завершается, то функция возвращает вызывающей стороне код ошибки.

Для соблюдения ограничений времени выполнения интерфейсы с проверкой ограничений вызывают соответствующий обработчик, который может просто завершиться или, как вариант, записать сообщение в stderr и/или аварийно завершить программу. Для выбора обработчика, который должен вызываться, можно использовать функцию set_constraint_handler_s; вдобавок вы можете заставить обработчик немедленно вернуться, как показано ниже:

```
int main(void) {
  constraint_handler_t oconstraint =
    set_constraint_handler_s(ignore_handler_s);
  get_y_or_n();
}
```

Если обработчик завершается, то функция, которая обнаружила нарушение ограничения времени выполнения и инициировала его вызов, сообщает о сбое вызывающей стороне с помощью своего возвращаемого значения.

Функции, интерфейс которых предполагает проверку выходов за границы массивов, обычно проверяют условия непосредственно при входе или по мере выполнения своих задач и сбора информации, достаточной для определения того, было ли нарушено ограничение времени выполнения. Условия, которые используются в функциях с проверкой выходов за границы массивов, приводили бы к неопределенному поведению в функциях из стандартной библиотеки C.

В реализациях есть обработчики ограничений, которые вызываются по умолчанию, если не использовать функцию `set_constraint_handler_s`. Они могут приводить к обычному или аварийному завершению программы, но разработчикам реализаций рекомендуется делать их поведение адекватным. Это, к примеру, позволяет компиляторам, которые обычно применяются в системах с повышенными требованиями к безопасности, не завершать программу аварийно по умолчанию. Если функция что-то возвращает, то вы должны проверять результаты ее работы, а не просто надеяться на то, что они окажутся корректными. Поведение, определяемое реализацией, можно переопределить, воспользовавшись функцией `set_constraint_handler_s` перед вызовом каких-либо функций с проверкой выходов за границы массивов или обратившись к механизму, который вызывает обработчик нарушения ограничений времени выполнения.

В приложении K предусмотрены функции `abort_handler_s` и `ignore_handler_s`, которые представляют две распространенные стратегии обработки ошибок. В зависимости от реализации обе эти функции могут и не быть обработчиками по умолчанию.

POSIX

POSIX тоже определяет несколько функций для работы со строками, таких как `strdup` и `strndup` (IEEE Std 1003.1:2018), которые представляют собой еще один набор строковых API для POSIX-совместимых платформ. Комитет во главе стандарта C опубликовал эти интерфейсы в техническом отчете 24731-2 (ISO/IEC TR 24731-2:2010), хотя в сам стандарт они пока не входят.

Чтобы исключить переполнение буфера, эти альтернативные функции используют динамически выделяемую память, реализуя модель, в которой

выделением памяти занимается *вызываемый* код, а освобождением — *вызывающий*. Каждая функция следит за выделением необходимого объема памяти (за исключением тех случаев, когда вызов `malloc` завершается неудачно). Функция `strdup`, к примеру, возвращает указатель на новую строку, которая содержит копию ее аргумента. Когда копия больше не нужна, ее память нужно освободить, передав возвращенный указатель функции `free` из стандартной библиотеки C.

В листинге 7.8 показан фрагмент кода, который использует функцию `strdup` для копирования строки, возвращенной вызовом `getenv`.

Листинг 7.8. Копирование строки с помощью функции strdup

```
const char *temp = getenv("TMP");
if (temp != NULL) {
  char *tmpvar = strdup(temp);
  if (tmpvar != NULL) {
    printf("TMP = %s.\n", tmpvar);
    free(tmpvar);
  }
}
```

Функция `getenv` из стандартной библиотеки C ищет в списке переменных среды, предоставленном средой исполнения, определенную строку (в данном примере это `"TMP"`). Строки в этом списке называются *переменными среды* и предоставляют дополнительный механизм для передачи процессу строковой информации. Они не имеют четко определенной кодировки, но обычно кодируются тем же методом, что и аргументы командной строки, `stdin` и `stdout`.

Возвращаемая строка (значение переменной) может быть перезаписана последующим вызовом функции `getenv`, поэтому переменные среды рекомендуется читать до создания каких-либо потоков выполнения, чтобы избежать потенциального состояния гонки. Если строка из этого списка может пригодиться позже, то ее следует скопировать и при необходимости использовать данную копию. Распространенный пример этого показан выше, в листинге 7.8.

Функция `strndup` является эквивалентом `strdup`, только копирует в выделенную память не больше `n + 1` байт (в то время как `strdup` копирует строку целиком) и следит за тем, чтобы новая строка всегда завершалась нуль-символом.

Эти POSIX-функции могут помочь избежать переполнения буфера за счет автоматического выделения места для итоговой строки, но, когда данная строка больше не нужна, вы должны сделать дополнительные вызовы `free`. Это значит, что после каждого использования `strdup` или, к примеру, `strndup` необходимо вызывать `free`, что может смутить программистов, которые лучше знакомы с поведением строковых функций из `<string.h>`.

Microsoft

В продуктах Microsoft реализовано большинство функций из стандартной библиотеки C, а также некоторые части стандарта POSIX. Но иногда эти реализации расходятся с требованиями соответствующего стандарта или имеют имена, конфликтующие с идентификаторами, зарезервированными в каком-то другом стандарте. В подобных случаях разработчики Microsoft часто добавляют перед именем функции подчеркивание. Например, в Windows нет POSIX-функции `strdup`, но вместо нее предусмотрена функция `_strdup`, которая ведет себя точно так же.

Библиотека Visual C++ включает экспериментальную реализацию функций с проверкой выхода за границы массивов. К сожалению, она не соответствует C11 и TR 24731-1, так как компания Microsoft решила не интегрировать в нее изменения, произошедшие в ходе процесса стандартизации этих API. Например, Visual C++ не предоставляет `set_constraint_handler_s`, поддерживая вместо нее более старый вызов с аналогичным поведением, но несовместимой сигнатурой:

```
_invalid_parameter_handler _set_invalid_parameter_handler(_invalid_
parameter_handler)
```

В продуктах Microsoft также отсутствуют функции `abort_handler_s` и `ignore_handler_s`, вызов `memset_s` (который не был определен в стандарте TR 24731-1) и макрос `RSIZE_MAX`. Кроме того, Visual C++ не считает пересечение буфера-источника и буфера-приемника нарушением ограничений времени выполнения; поведение в таком случае просто считается неопределенным. В моей исследовательской работе для NCC Group, называющейся Bounds-Checking Interfaces: Field Experience and Future Directions, можно найти дополнительную информацию обо всех аспектах интерфейсов с проверкой ограничений, в том числе и о реализации от Microsoft (Сикорд, 2019).

Резюме

В этой главе вы познакомились с кодировками, такими как ASCII и Unicode, и типами данных `char`, `int`, `wchar_t` и т. д., которые используются для представления символов в программах на языке C. Затем мы прошлись по инструментам для преобразования символов, включая функции стандартной библиотеки C, `libiconv` и API Windows.

Помимо символов, мы рассмотрели строки, устаревшие функции и функции, интерфейс которых предполагает проверку выходов за границы массивов, определенные в стандартной библиотеке C и предназначенные для работы со строками, а также функции, относящиеся к POSIX и продуктам Microsoft.

В следующей главе мы обсудим ввод/вывод.

8 Ввод/вывод

В этой главе вы научитесь выполнять операции ввода/вывода для чтения и записи данных в терминал и файловую систему. К вводу/выводу относятся все способы, с помощью которых информация поступает в программу и выходит из нее. Без этого ваши программы были бы бесполезными. Мы рассмотрим методики на основе стандартных потоков C и файловых дескрипторов POSIX. Начнем наше обсуждение с текстовых и бинарных потоков, доступных в языке C, а затем поговорим о разных способах открытия и закрытия файлов с помощью стандартной библиотеки C и функций POSIX. Вслед за этим речь пойдет о чтении и записи символов, строк (в файлах) и форматированного текста, а также о чтении из двоичных потоков и записи в них. Мы затронем такие темы, как буферизация потоков, ориентация потоков и указание позиции в файлах.

В эту главу не вошло множество других устройств и интерфейсов ввода/вывода (таких как `ioctl`).

Стандартные потоки ввода/вывода

В спецификации языка C определены потоки, предназначенные для взаимодействия с терминалами и файлами, хранящимися в поддерживаемых, структурированных устройствах хранения. *Поток (stream)* — это унифицированная абстракция для работы с файлами и устройствами, потребляющими или производящими последовательные данные, — сокетами, клавиатурами, USB-портами и принтерами.

Для представления потоков в языке C используется непрозрачный тип данных `FILE`. Объект `FILE` хранит внутреннюю информацию о соединении с соответствующим файлом, включая указатель на позицию в файле, сведения о буферизации, индикатор ошибок и индикатор конца файла. Вы никогда не должны выделять объект `FILE` самостоятельно. Функции стандартной библиотеки C работают с объектами типа `FILE *` (то есть с указателями на `FILE`). В связи с этим потоки нередко называют *указателями на файлы*.

В спецификации C предусмотрен обширный API для работы с потоками, `<stdio.h>`; мы исследуем его позже в данной главе. Но поскольку эти функции должны работать с разнообразными устройствами и файловыми системами на множестве платформ, они являются очень абстрактными, что делает их пригодными лишь для простейших задач.

Например, в стандарте C нет такого понятия, как каталог, поскольку стандарт должен быть совместим с неиерархическими файловыми системами. В нем есть несколько упоминаний механизмов, характерных лишь для некоторых ФС, включая права доступа к файлам и блокирование. Однако в спецификациях функций зачастую отмечается, что определенные аспекты поведения присутствуют «в той степени, в которой они поддерживаются конкретной системой»; это значит, что они доступны только там, где реализована их поддержка.

В результате для выполнения ввода/вывода в реальных приложениях вам, скорее всего, придется задействовать менее переносимые API, которые предоставляют POSIX, Windows и другие платформы. Многие приложения используют собственные API для безопасного и кроссплатформенного ввода/вывода, основанные на механизмах, доступных в той или иной системе.

Буферизация потоков

Буферизация — это процесс временного хранения в основной памяти данных, которыми обмениваются программа и устройство или файл. Буферизация улучшает пропускную способность операций ввода/вывода, для которых характерна высокая латентность. Аналогично, когда программа делает запрос на запись в блочное устройство, такое как диск, драйвер может кэшировать данные в памяти; когда их накопится достаточно для

заполнения одного или нескольких блоков, они будут записаны на диск за одну операцию, что улучшит пропускную способность. Эта процедура называется *сбросом* выходного буфера.

Потоки, как и драйверы устройств, нередко используют собственные буферы ввода/вывода. Обычно поток выделяет по одному входному и выходному буферу для каждого файла, который программа хочет прочитать или записать.

Поток может находиться в одном из трех состояний.

- *Небуферизованный* — символы должны браться из источника или сохраняться по месту назначения как можно раньше. Небуферизованными могут быть потоки, предназначенные для сообщений об ошибках или логирования.
- *Полностью буферизованный* — символы должны передаваться в среду выполнения или из нее в виде блоков при заполнении буфера. Полностью буферизованные потоки обычно используют для файлового ввода/вывода, чтобы оптимизировать пропускную способность.
- *Строчно-буферизованный* — символы должны передаваться в среду выполнения или из нее в виде блоков при обнаружении символа перевода строки. Строчно-буферизованные потоки соединяются с интерактивными устройствами, такими как терминалы.

В следующем подразделе мы познакомимся со стандартными (predefined) потоками и посмотрим, как они буферизуются.

Потоки из стандартной библиотеки

У вашей программы есть три *стандартных текстовых потока*, которые открываются и становятся доступными при ее запуске. Все они объявлены в `<stdio.h>`:

```
extern FILE * stdin;  // стандартный поток ввода
extern FILE * stdout; // стандартный поток вывода
extern FILE * stderr; // стандартный поток ошибок
```

Стандартный поток вывода (`stdout`) — то место, куда принято записывать вывод программы. Он обычно связан с терминалом, в котором программа

была запущена, но его можно перенаправить в выходной файл или другой поток, как показано ниже:

```
$ echo fred
fred
$ echo fred > tempfile
$ cat tempfile
fred
```

Здесь вывод команды `echo` перенаправляется в `tempfile`.

Стандартный поток ввода (`stdin`) — это общепринятый источник ввода программы. По умолчанию он связан с клавиатурой, но к нему можно перенаправить входной файл, например, с помощью таких команд:

```
$ echo "one two three four five six seven" > fred
$ wc < fred
1 7 34
```

Содержимое файла `fred` перенаправляется в `stdin` команды `wc`, которая выводит количество символов перевода строки (`1`), слов (`7`) и байтов (`34`) внутри `fred`.

Стандартный поток ошибок (`stderr`) предназначен для записи диагностического вывода. В отличие от `stdin` и `stdout`, которые полностью буферизуются лишь в том случае, если поток не указывает на интерактивное устройство, `stderr` при первом открытии не является полностью буферизованным; благодаря этому сообщения об ошибках можно просматривать без лишних задержек.

На рис. 8.1 показаны потоки `stdin`, `stdout` и `stderr`, соединенные с клавиатурой и экраном пользовательского терминала.

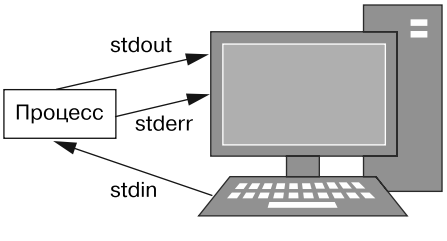

Рис. 8.1. Стандартные потоки, соединенные с каналами ввода/вывода

Выходной поток одной программы может быть перенаправлен во входной поток другой с помощью каналов POSIX. Многие операционные системы позволяют соединять приложения в цепочку путем разделения команд символом вертикальной черты (|):

```
$ echo "Hello Robert" | sed "s/Hello/Hi/" | sed "s/Robert/robot/"
Hi robot
```

Ориентация потоков

Каждый поток имеет *ориентацию*, которая определяет, какие символы он содержит: узкие или широкие. После связывания потока с файлом, но до выполнения с ним каких-либо операций у него нет ориентации. Как только к потоку применяется широкосимвольная функция ввода/вывода, он становится *широкоориентированным*. Точно так же в случае применения к неориентированному потоку байтовой функции ввода/вывода он становится *байтоориентированным*. Многобайтные символьные последовательности или узкие символы, которые можно представить в виде объектов типа `char` (который согласно требованиям стандарта C должен занимать один байт), доступны для записи в байтоориентированный поток.

Вы можете сбросить ориентацию потока с помощью функции `fwide` или за счет закрытия и повторного открытия файла. Применение байтовых функций ввода/вывода к широкоориентированному потоку или широкосимвольных функций ввода/вывода к байтоориентированному потоку приводит к неопределенному поведению. Никогда не смешивайте узкие символы, широкие символы и двоичные данные в одном файле.

Во время запуска программы все три стандартных потока (`stderr`, `stdin` и `stdout`) являются неориентированными.

Текстовые и двоичные потоки

Стандарт C поддерживает как текстовые, так и двоичные потоки. *Текстовый поток* — это упорядоченная последовательность символов, объединенных в строки, каждая из которых состоит из произвольного количества символов (начиная с нуля) с последовательностью символов перевода строки в конце. В Unix-подобных системах перевод строки можно обозначить с помощью символа `\n`. Большинство программ для Microsoft

Windows используют возврат каретки `\r`, за которым идет символ перевода строки `\n`.

Поскольку в разных системах перевод строки принято обозначать по-разному, при переносе текстовых файлов их содержимое может отображаться неправильно. Если создать текстовый файл в одной из популярных программ в Unix-подобной операционной системе и затем открыть его в старой программе для Microsoft Windows, которая не интерпретирует одиночные символы `\r` или `\n` как перевод строки, он будет выглядеть как одна длинная строчка.

Двоичный (бинарный) поток — это упорядоченная последовательность произвольных двоичных данных. В отдельно взятой реализации данные, прочитанные из потока, идентичны данным, которые были ранее записаны в тот же поток. Хотя в конец этих потоков может добавляться разное количество нулевых байтов в зависимости от реализации.

Двоичные потоки всегда являются более мощными и предсказуемыми по сравнению с текстовыми. Но если вам нужно читать или записывать обычные текстовые файлы, совместимые с другими программами для работы с текстом, то проще всего будет воспользоваться текстовыми потоками.

Открытие и создание файлов

Когда вы открываете или создаете файл, он соединяется с потоком. Для открытия и создания файлов предусмотрены функции, о которых я расскажу ниже.

Функция fopen

Функция `fopen` открывает файл, имя которого передано ей в виде указателя на строку `filename`, и затем соединяет его с потоком. Если такой файл не существует, то `fopen` его создаст:

```
FILE *fopen(
  const char * restrict filename,
  const char * restrict mode
);
```

Аргумент `mode` указывает на одну из строк, перечисленных в табл. 8.1, чтобы определить режим открытия файла.

Таблица 8.1. Допустимые режимы открытия файла

Режим	Описание
r	Открытие существующего текстового файла для чтения
w	Усечение до нулевой длины или открытие текстового файла для записи
a	Добавление, открытие или создание текстового файла для записи в его конец
rb	Открытие существующего двоичного файла для чтения
wb	Усечение до нулевой длины или открытие двоичного файла для записи
ab	Добавление, открытие или создание двоичного файла для записи в конец
r+	Открытие существующего текстового файла для чтения и записи
w+	Усечение до нулевой длины или открытие текстового файла для чтения и записи
a+	Добавление, открытие или создание текстового файла для обновления, записи в его конец
r+b или rb+	Открытие существующего двоичного файла для обновления (чтения и записи)
w+b или wb+	Усечение до нулевой длины или открытие двоичного файла для чтения и записи
a+b или ab+	Добавление, открытие или создание двоичного файла для обновления, записи в его конец

Открытие файла в режиме «для чтения» (путем передачи `r` в качестве первого символа в аргументе `mode`) завершается неудачей, если данный файл не существует или его нельзя прочитать. Открытие файла в режиме присоединения (путем передачи `a` в качестве первого символа в аргументе `mode`) приводит к тому, что любые данные, которые записываются в файл, добавляются в его конец. В ряде реализаций открытие двоичного файла в режиме присоединения путем передачи `b` в качестве второго или третьего символа в аргументе `mode` может сначала установить текущую позицию в потоке за пределы последнего записанного фрагмента данных; это объясняется заполнением потока нуль-символами.

Чтобы открыть файл в режиме обновления, в качестве второго или третьего символа в аргументе `mode` можно указать `+`. Это позволит выполнять с соответствующим потоком операции чтения и записи. В некоторых реализациях открытие (или создание) текстового файла в данном режиме может привести к открытию (или созданию) двоичного потока.

В спецификации C11 появился *эксклюзивный режим* для чтения и записи двоичных и текстовых файлов. Он представлен в табл. 8.2.

Таблица 8.2. Допустимые режимы открытия файла, появившиеся в C11

Режим	Описание
wx	Создание эксклюзивного текстового файла для записи
wbx	Создание эксклюзивного двоичного файла для записи
w+x	Создание эксклюзивного текстового файла для чтения и записи
w+bx или wb+x	Создание эксклюзивного двоичного файла для чтения и записи

Открытие файла в эксклюзивном режиме (путем передачи `x` в качестве последнего символа в аргументе `mode`) завершается неудачей, если файл уже существует или его не удается создать. В противном случае файл создается с эксклюзивным (или *неразделяемым*) доступом в той мере, в которой это поддерживается системой.

В завершение стоит отметить, что объект `FILE` нельзя копировать. Например, в следующей программе может произойти сбой, поскольку в вызове `fputs` используется копия содержимого `stdout`:

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  FILE my_stdout = *stdout;
  if (fputs("Hello, World!\n", &my_stdout) == EOF) {
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

Эта программа имеет неопределенное поведение и при запуске обычно аварийно завершается.

Функция open из стандарта POSIX

В POSIX-системах функция `open` (IEEE Std 1003.1:2018) устанавливает соединение между файлом с идентификатором `path` и значением, которое называют *файловым дескриптором*:

```
int open(const char *path, int oflag, ...);
```

Файловый дескриптор — неотрицательное целое число, обозначающее структуру, которая представляет файл (известную как *описание открытого файла*). Файловый дескриптор, возвращенный функцией `open`, является наименьшим целым числом, которое еще не возвращалось из `open` и не передавалось в `close`; оно уникальное в рамках вызывающего процесса. Файловый дескриптор используют другие функции ввода/вывода для обращения к тому же файлу. Функция `open` устанавливает смещение внутри файла, которое служит для обозначения текущей позиции, в его начало.

Значение параметра `oflag` устанавливает *режим доступа* для описания открытого файла, который определяет, был ли файл открыт для чтения или записи (или того и другого). Значение `oflag` конструируется как битовое ИЛИ одного из режимов доступа и любого сочетания флагов состояния. В значении `oflag` необходимо указать ровно один из следующих режимов:

- `O_EXEC` — открытие только для выполнения (для файлов, не являющихся каталогами);
- `O_RDONLY` — открытие только для чтения;
- `O_RDWR` — открытие для чтения и записи;
- `O_SEARCH` — открытие каталога только для поиска;
- `O_WRONLY` — открытие только для записи.

Значение параметра `oflag` также устанавливает *флаги состояния файла*, которые определяют поведение функции `open` и влияют на выполнение файловых операций. В число этих флагов входят:

- `O_APPEND` — устанавливает смещение внутри файла в его конец перед каждой операцией записи;
- `O_TRUNC` — усекает длину до 0;
- `O_CREAT` — создает файл;

- `O_EXCL` — если флаг `O_CREAT` тоже установлен, а файл уже существует, то приводит к неудачному открытию.

Функция `open` имеет переменное количество аргументов. Значение аргумента, который идет за `oflag`, определяет биты файлового режима (права доступа к создаваемому файлу) и имеет тип `mode_t`.

В листинге 8.1 показан пример использования функции `open` в целях открытия файла с возможностью записи со стороны его владельца.

Листинг 8.1. Открытие файла его владельцем, только для записи

```
#include <fcntl.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
//---snip---
int fd;
❶ mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
const char *pathname = "/tmp/file";
//---snip---
if ((fd = open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode)❷) == -1)
{
  fprintf(stderr, "Can't open %s.\n", pathname);
  exit(1);
}
//---snip---
```

Мы создаем флаг `mode` ❶, который является битовым включающим ИЛИ перечисленных ниже битов, описывающих права доступа:

- `S_IRUSR` — бит, разрешающий владельцу читать файл;
- `S_IWUSR` — бит, разрешающий владельцу записывать файл;
- `S_IRGRP` — бит, разрешающий группе владельцев читать файл;
- `S_IROTH` — бит, разрешающий читать файл другим пользователям.

Вызов `open` ❷ принимает несколько аргументов, включая путь к файлу, флаг `oflag` и режим. Режим доступа к файлу равен `O_WRONLY`; это значит, что он открывается только для записи. Флаг состояния `O_CREAT` заставляет функцию `open` создать новый файл; флаг состояния `O_TRUNC` говорит о том, что если существующий файл открыт успешно, то его содержимое нужно удалить, но сохранить при этом его идентификатор.

Если файл удается открыть, то функция `open` возвращает неотрицательное целое число, представляющее файловый дескриптор. Если нет, то она возвращает `–1` и устанавливает `errno` для оповещения об ошибке[1]. Код в листинге 8.1 проверяет, равно ли значение `–1`, записывает диагностическое сообщение в стандартный поток `stderr`, если произошла ошибка, и затем завершается.

Помимо `open`, POSIX предлагает другие полезные функции для работы с файловыми дескрипторами. Например, `fileno` возвращает файловый дескриптор, связанный с указателем на существующий файл, а `fdopen` создает новый указатель на файловый поток из текущего файлового дескриптора. API POSIX, доступные через файловые дескрипторы, позволяют использовать такие возможности файловых систем, как каталоги, права доступа, символические и жесткие ссылки, которые обычно нельзя получить с помощью интерфейсов, работающих с указателями на файлы.

Закрытие файлов

Открытие файла выделяет ресурсы. Если вы постоянно открываете файлы и никогда их не закрываете, то у вашего процесса рано или поздно закончатся доступные файловые дескрипторы или идентификаторы, после чего все последующие попытки открытия будут неудачными. В связи с этим после завершения работы с файлами их необходимо закрывать.

Функция fclose

Функция `fclose` из стандартной библиотеки C закрывает файл:

```
int fclose(FILE *stream);
```

Любые буферизованные данные, которые не успели записаться в поток, записываются средой выполнения в файл. Любые непрочитанные буферизованные данные сбрасываются.

Функция `fclose` может завершиться неудачей. Например, при записи оставшегося буферизованного вывода она может вернуть ошибку из-за нехватки места на диске. При использовании протокола NFS (Network File System — сетевая файловая система) закрытие файла может спровоцировать сбой, даже если вы знаете, что его буфер пустой. Несмотря

[1] Это стандартная глобальная переменная, хранящая код последней ошибки. — *Примеч. науч. ред.*

на возможные сбои, восстановление зачастую невозможно, поэтому программисты обычно игнорируют ошибки, которые возвращает `fclose`. Общепринятый ответ на невозможность закрыть файл состоит в принудительном завершении процесса или усечении файла таким образом, чтобы при следующем чтении его содержимое было осмысленным.

После закрытия файла значение указателя на связанный с ним объект `FILE` становится неопределенным. Возможность существования файлов с нулевой длиной (в которые выходной поток не записал никаких данных) зависит от реализации.

Вы можете повторно открыть закрытый файл в той же или другой программе, и его содержимое может быть восстановлено или изменено. При завершении функции `main` или вызове `exit` все открытые файлы закрываются (и все выходные потоки сбрасываются на диск) перед завершением программы.

Если программа завершается каким-то другим образом (например, с помощью функции `abort`), то корректного закрытия файлов может не произойти. В этом случае буферизованные данные, которые еще не были записаны на диск, могут быть утеряны.

Функция close из стандарта POSIX

В POSIX-системах можно использовать функцию `close` для освобождения файлового дескриптора, указанного с помощью `fildes`:

```
int close(int fildes);
```

Если во время закрытия файла происходит ошибка ввода/вывода, связанная с операциями чтения или записи, то `close` может вернуть `–1` и установить `errno` значение `EIO`. В случае возвращения этой ошибки состояние `fildes` становится неопределенным — то есть вы больше не сможете читать или записывать данные в этот дескриптор или пытаться закрыть его еще раз.

После закрытия файла его дескриптор перестает существовать, поскольку целое число, которое его представляет, больше не ссылается на файл. Файлы также закрываются при завершении работы процесса, который владеет их потоками.

Если файл был открыт с помощью `fopen`, то для его закрытия необходимо использовать `fclose`; функция `close` используется для закрытия файлов,

открытых с помощью `open` (если только ее дескриптор не был передан функции `fdopen`; в этом случае для закрытия файла нужно вызвать `fclose`).

Чтение и запись символов и строк

В стандарте C предусмотрены функции для чтения и записи отдельных символов и строк.

Большинство функций для работы с байтовыми потоками имеют аналоги, которые принимают широкие символы (`wchar_t`) или широкосимвольные строки вместо узких символов (`char`) и строк соответственно (табл. 8.3). Функции для работы с потоками байтов определены в `<stdio.h>`, а широкосимвольные потоковые функции — в `<wchar.h>`. Последние работают с теми же потоками (такими как `stdout`).

Таблица 8.3. Узко- и широкосимвольные функции ввода/вывода

char	wchar_t	Описание
fgetc	fgetwc	Читает символ из потока
getc	getwc	Читает символ из потока (функция зачастую является макросом)
getchar	getwchar	Читает символ из stdin
fgets	fgetws	Читает строчку из потока
fputc	fputwc	Записывает символ в поток
putc	putwc	Записывает символ в поток (зачастую является макросом)
fputs	fputws	Записывает строку в поток
putchar	putwchar	Записывает символ в stdout
puts	—	Записывает строку в stdout
ungetc	ungetwc	Возвращает символ в поток
scanf	wscanf	Читает форматированный символьный ввод из stdin
fscanf	fwscanf	Читает форматированный символьный ввод из потока
sscanf	swscanf	Читает форматированный символьный ввод из буфера
printf	wprintf	Записывает форматированный символьный вывод в stdout
fprintf	fwprintf	Записывает форматированный символьный вывод в поток
sprintf	swprintf	Записывает форматированный символьный вывод в буфер
snprintf	—	То же самое, что sprintf с усечением. Функция swprintf также принимает в качестве аргумента длину, но интерпретирует ее не так, как snprintf

В этой главе мы обсудим только функции для работы с байтовыми потоками. Их широкосимвольных аналогов лучше избегать и по возможности использовать вместо них исключительно методы кодирования UTF-8, так как последние менее подвержены ошибкам и уязвимостям безопасности.

Функция `fputc` приводит символ `c` к типу `unsigned char` и записывает его в поток `stream`:

```
int fputc(int c, FILE *stream);
```

Если возникает ошибка записи, то она возвращает `EOF`; в противном случае возвращается записанный символ.

Функция `putc` ничем не отличается от `fputs`, только в большинстве сред реализована в виде макроса:

```
int putc(int c, FILE *stream);
```

Если функция `putc` является макросом, то может вычислять свои аргументы повторно, поэтому никогда не передавайте ей выражения с побочными эффектами.

Функция `fputc` обычно более безопасна. Подробности об этом ищите в правиле FIO41-C стандарта CERT C (не передавайте вызовам `getc()`, `putc()`, `getwc()` и `putwc()` аргумент `stream` с побочными эффектами).

Функция `putchar` является эквивалентом `putc`, только в качестве аргумента `stream` использует `stdout`.

Функция `fputs` записывает строку `s` в поток `stream`:

```
int fputs(const char * restrict s, FILE * restrict stream);
```

Эта функция записывает только содержимое строки `s`, без нулевого байта, и не делает перевод строки. В случае ошибки `fputs` возвращает `EOF`. Если запись прошла успешно, то возвращается неотрицательное значение. Например, следующие операторы выводят текст `I am Groot` с переводом строки в конце:

```
fputs("I ", stdout);
fputs("am ", stdout);
fputs("Groot\n", stdout);
```

Функция puts записывает строку s в stdout и выполняет перевод строки в конце:

```
int puts(const char *s);
```

Функция puts принимает лишь один аргумент и поэтому является самым удобным средством вывода простых сообщений. Например:

```
puts("This is a message.");
```

Функция fgetc читает из потока следующий символ как unsigned char и возвращает его значение, приведенное к типу int:

```
int fgetc(FILE *stream);
```

В случае выполнения условия конца файла или возникновения ошибки эта функция возвращает EOF.

Как вы, наверное, помните, функция gets читает символы из stdin и записывает их в символьный массив, пока не достигает перевода строки или EOF. Она является небезопасной по своей природе. В C99 ее признали устаревшей и не включили в C11. *Никогда ее не используйте.* Если вам нужно прочитать строку из stdin, то попробуйте сделать это с помощью fgets. Функция fgets может прочитать из потока в символьный массив максимум «указанное количество» минус один (чтобы оставить место для нуль-символа).

Сброс потока на диск

Как уже описывалось ранее в этой главе, потоки могут быть частично или полностью буферизованными. Это значит, что данные, которые вы считаете записанными, могут еще не быть в распоряжении среды выполнения. В частности, могут возникнуть проблемы, если программа завершается преждевременно. Функция fflush передает любые данные, которые не успели записаться в заданный поток, среде выполнения, чтобы она сохранила их в файл:

```
int fflush(FILE *stream);
```

Если последней операцией с потоком был ввод, то поведение не определено. Если поток представляет собой нулевой указатель, то функция fflush выполняет сброс на диск для всех потоков. Если вы не собираетесь этого

делать, то убедитесь в том, что указатель на файл, который передается в `fflush`, не является нулевым. В случае ошибки записи функция `fflush` устанавливает код ошибки для потока и возвращает `EOF`; в противном случае возвращается нуль.

Установка позиции в файле

Потоки с произвольным доступом (такие как файлы на диске, но не терминал) хранят *текущую позицию* в файле. Она определяет, какой участок файла в настоящее время читается или записывается потоком.

В момент открытия файла текущая позиция находится в его начале. Вы можете переместить ее куда угодно, чтобы прочитать или записать любую часть файла. Функция `ftell` получает значение текущей позиции в файле, а `fseek` позволяет ее установить. Для представления смещения эти функции используют `long int`, в связи с чем смещение может быть не больше, чем размер данного типа. Применение функций `ftell` и `fseek` показано в листинге 8.2.

Листинг 8.2. Использование функций ftell и fseek

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  FILE *fp = fopen("fred.txt", "r");
  if (fp == NULL) {
    fputs("Cannot open fred.txt file\n", stderr);
    return EXIT_FAILURE;
  }
  if (fseek(fp, 0, SEEK_END) != 0) {
    fputs("Seek to end of file failed\n", stderr);
    return EXIT_FAILURE;
  }
  long int fpi = ftell(fp);
  if (fpi == -1L) {
    perror("Tell");
    return EXIT_FAILURE;
  }
  printf("file position = %ld\n", fpi);
  if (fclose(fp) == EOF) {
    fputs("Failed to close file\n", stderr);
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

Эта программа открывает файл `fred.txt` и вызывает `fseek`, чтобы установить текущую позицию в его конец (обозначенный как `SEEK_END`). Функция `ftell` возвращает значение текущей позиции в файле для заданного потока в виде `long int`. Программа выводит данное значение и завершается. В конце мы закрываем файл, на который ссылается указатель `fp`. Чтобы сделать свой код надежным, не забывайте проводить проверку ошибок. В частности, сбои, происходящие с файловым вводом/выводом, могут быть вызваны всевозможными причинами. В случае ошибки функция `fopen` возвращает нулевой указатель. Функция `fseek` возвращает ненулевое значение только для запроса, который не удается удовлетворить. В ответ на сбой функция `ftell` возвращает `–1L` и сохраняет в `errno` значение, определяемое реализацией. Если возвращаемое из `ftell` значение равно `–1L`, то мы используем функцию `perror`, чтобы вывести указанную нами строку, Tell, двоеточие (`:`), подходящее сообщение, которое соответствует значению, хранящемуся в `errno`, и символ перевода строки в конце. В случае обнаружения каких-либо ошибок функция `fclose` возвращает `EOF`. Один из неблаговидных аспектов стандартной библиотеки C, продемонстрированный этой короткой программой, состоит в том, что каждая функция пытается сообщать об ошибках по-своему, поэтому, чтобы узнать, как проверять ошибки в том или ином случае, нужно сверяться с документацией.

Новейшие функции `fgetpos` и `fsetpos` используют для представления смещений типа `fpos_t`, который может описывать смещения произвольного размера. Это значит, что их можно применять для работы со сколь угодно большими файлами. У широкоориентированного потока есть объект `mbstate_t`, который хранит его текущее состояние разбора. Успешный вызов `fgetpos` сохраняет эту многобайтную информацию о состоянии в рамках объекта `fpos_t`. Следующий вызов `fsetpos` с тем же значением `fpos_t` восстанавливает состояние разбора и позицию внутри управляемого потока. Объект `fpos_t` можно преобразовать в целочисленное смещение (байтовое или символьное) потока только косвенно, путем последовательного вызова `fsetpos` и `ftell`. В листинге 8.3 показана короткая программа, которая демонстрирует применение функций `fgetpos` и `fsetpos`.

Листинг 8.3. Использование функций fgetpos и fsetpos

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(void) {
  FILE *fp = fopen("fred.txt", "w+");
  if (fp == NULL) {
    fputs("Cannot open fred.txt file\n", stderr);
    return EXIT_FAILURE;
  }
  fpos_t pos;
  if (fgetpos(fp, &pos) != 0) {
    perror("get position");
    return EXIT_FAILURE;
  }
  if (fputs("abcdefghijklmnopqrstuvwxyz", fp) == EOF) {
      fputs("Cannot write to fred.txt file\n", stderr);
  }
  if (fsetpos(fp, &pos) != 0) {
    perror("set position");
    return EXIT_FAILURE;
  }
  long int fpi = ftell(fp);
  if (fpi == -1L) {
    perror("seek");
    return EXIT_FAILURE;
  }
  printf("file position = %ld\n", fpi);
  if (fputs("0123456789", fp) == EOF) {
    fputs("Cannot write to fred.txt file\n", stderr);
  }
  if (fclose(fp) == EOF) {
    fputs("Failed to close file\n", stderr);
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

Эта программа открывает файл `fred.txt` для записи и затем вызывает `fgetpos`, чтобы получить текущую позицию, которая сохраняется в `pos`. Затем мы записываем в файл определенный текст и вызываем `fsetpos`, чтобы вернуть текущую позицию к значению, хранящемуся в `pos`. Затем мы можем использовать функцию `ftell`, чтобы извлечь и вывести текущую позицию в файле, которая должна быть равна 0. В результате выполнения этой программы файл `fred.txt` должен содержать следующий текст:

```
0123456789klmnopqrstuvwxyz
```

Вы не можете записать в поток какие-то данные и затем прочитать их оттуда, не прибегая к промежуточному вызову функции `fflush`, которая сохранит все, что не успело записаться, или функции для изменения

текущей позиции (`fseek`, `fsetpos` либо `rewind`). Вы также не можете выполнить чтение из потока с последующей записью в него, не совершив промежуточный вызов одной из функций, которая изменяет текущую позицию в файле.

Функция `rewind` устанавливает текущую позицию файла в его начало:

```
void rewind(FILE *stream);
```

Функция `rewind` эквивалентна последовательному вызову `fseek` и `clearerr` (для очистки индикатора ошибки потока):

```
fseek(stream, 0L, SEEK_SET);
clearerr(stream);
```

Не пытайтесь с помощью функций изменять текущую позицию в файлах, открытых в режиме присоединения, так как многие системы либо не обновляют ее в этом режиме либо принудительно сбрасывают ее в конец файла во время записи. API, которые работают с позицией файла, сохраняют ее при последующих операциях чтения, записи и изменения позиции. В POSIX и Windows доступны API, которые никогда не используют текущую позицию; для них всегда нужно указывать смещение, по которому будет выполняться ввод/вывод.

Удаление и переименование файлов

Стандартная библиотека C предоставляет функции `remove` и `rename` для удаления и соответственно переименования файлов:

```
int remove(const char *filename);
int rename(const char *old, const char *new);
```

В POSIX функция для удаления файлов называется `unlink`:

```
int unlink(const char *path);
```

Функция `unlink` имеет более четкую семантику, поскольку относится к файловым системам POSIX. Для переименования в POSIX тоже используется `rename`. Здесь, как и в Windows, у файла может быть сколько угодно ссылок, включая жесткие ссылки и дескрипторы открытого файла. Функция `unlink` всегда убирает запись о файле из каталога, но его удаление происходит только после того, как на него больше не остается ссылок.

Но даже в этом случае содержимое файла может и дальше храниться на постоянной основе.

В системах POSIX функция `remove` ведет себя так же, как `unlink`, но в других ОС ее поведение может отличаться.

Использование временных файлов

Временные файлы часто используются в качестве механизма межпроцессного взаимодействия или для освобождения оперативной памяти за счет временного сохранения информации на диск. Например, один процесс может записывать во временный файл, из которого читает другой процесс. Такие файлы обычно создаются во временном каталоге с помощью таких функций, как `tmpfile` и `tmpnam`, из стандартной библиотеки C или `mkstemp` из состава POSIX.

Временные каталоги могут быть либо глобальными, либо пользовательскими. В Unix и Linux для задания местоположения глобальных временных каталогов используется переменная среды `TMPDIR`, которая обычно содержит пути `/tmp` и `/var/tmp`. В Linux временные пользовательские каталоги, как правило, определяются переменной среды `$XDG_RUNTIME_DIR`, которая чаще всего содержит значение `/run/user/$uid`. В Windows временные пользовательские каталоги можно найти в разделе `AppData` профиля пользователя; обычно это `C:\Users\Имя пользователя\AppData\Local\Temp` (`%USERPROFILE%\AppData\Local\Temp`). Глобальный временный каталог в Windows определяется одной из трех переменных среды: `TMP`, `TEMP` или `USERPROFILE`. В системной папке `C:\Windows\Temp` Windows хранит свои временные файлы.

По соображениям безопасности каждому пользователю лучше работать с собственным временным каталогом, так как применение глобальных временных каталогов часто приводит к уязвимостям. Самая безопасная функция для создания временных файлов — `mkstemp` из состава POSIX. Но, поскольку безопасная реализация доступа к файлам в разделяемых каталогах может оказаться сложной или даже непосильной задачей, я рекомендую отказаться от временных файлов для межпроцессного взаимодействия и вместо этого выполнять его с помощью сокетов, разделяемой памяти или других механизмов, предназначенных для этой цели.

Чтение потоков форматированного текста

В этом разделе я продемонстрирую использование функции `fscanf` для чтения форматированного ввода. Она является входным аналогом функции `fprintf`, с которой вы познакомились еще в главе 1, и имеет следующую сигнатуру:

```
int fscanf(FILE * restrict stream, const char * restrict format, ...);
```

Функция `fscanf` читает ввод из потока, на который указывает `stream`, и с помощью строки `format` определяет, сколько аргументов нужно ожидать, какие у них типы и как их следует преобразовать для присваивания. Последующие аргументы являются указателями на объекты, которым присваивается преобразованный ввод. Если количество переданных аргументов меньше того, которое указано в строке `format`, то поведение программы не определено. Если же аргументов больше, то те из них, которые оказались лишними, вычисляются, но в остальном игнорируются. Функция `fscanf` имеет богатые возможности, и здесь мы затронем лишь небольшую их часть. Больше информации можно найти в стандарте C.

Чтобы продемонстрировать использование `fscanf` и некоторых других функций ввода/вывода, мы напишем программу, которая читает файл `signals.txt`, показанный в листинге 8.4, и построчно его выводит. В каждой строчке файла содержится:

- номер сигнала (небольшое положительное целое число);
- ID сигнала (небольшая строка длиной не более шести алфавитно-цифровых символов);
- короткая строка с описанием сигнала.

Поля разделены пробельными символами. Исключение составляет поле с описанием, которое может содержать пробелы; оно отделено переводом строки.

Листинг 8.4. Файл signals.txt

```
1 HUP Hangup
2 INT Interrupt
3 QUIT Quit
4 ILL Illegal instruction
5 TRAP Trace trap
```

```
6 ABRT Abort
7 EMT EMT trap
8 FPE Floating-point exception
```

В листинге 8.5 показана программа `signals`, которая читает этот файл и выводит каждую его строчку.

Листинг 8.5. Программа signals

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

int main(void) {
  int status = EXIT_SUCCESS;
  FILE *in;

  struct sigrecord {
      int signum;
      char signame[10];
      char sigdesc[100];
❶  } sigrec;

  if ((in = fopen("signals.txt", "r")) == NULL) {
    fputs("Cannot open signals.txt file\n", stderr);
    return EXIT_FAILURE;
  }

  do {
❷   int n = fscanf(in, "%d%9s%*[ \t]%99[^\n]",
      &sigrec.signum, sigrec.signame, sigrec.sigdesc
    );
    if (n == 3) {
      printf(
        "Signal\n number = %d\n name = %s\n description = %s\n\n",
        sigrec.signum, sigrec.signame, sigrec.sigdesc
      );
    }
    else if (n != EOF) {
      fputs("Failed to match signum, signame or sigdesc\n", stderr);
      status = EXIT_FAILURE;
      break;
    }
    else {
      break;
    }
❸  } while (1);

❹  if (fclose(in) == EOF) {
    fputs("Failed to close file\n", stderr);
```

```
    status = EXIT_FAILURE;
  }
  return status;
}
```

Мы определили несколько переменных в функции `main`, включая структуру `sigrec` ❶, в которой будет храниться информация о сигналах, найденная в каждой строчке файла. Эта структура имеет три члена: `signum` типа `int` для хранения номера сигнала, массив `signame` типа `char` для хранения ID сигнала и массив `sigdesc` типа `char`, в котором будет храниться описание сигнала. Оба массива имеют фиксированный размер, которого должно быть достаточно для чтения строк из файла. Если прочитанная строка оказывается слишком длинной и не помещается в массив, то программа будет считать это ошибкой.

Вызов `fscanf` ❷ считывает из файла каждую входную строчку. Он находится внутри бесконечного цикла `do...while(1)` ❸, из которого нужно выйти, чтобы завершить программу. Мы присваиваем значение, возвращаемое функцией `fscanf`, локальной переменной `n`. Если до проведения первого преобразования происходит сбой ввода, то функция `fscanf` возвращает `EOF`. В противном случае возвращается количество присвоенных входных элементов, которое может быть меньше указанного или даже равно нулю, если ошибка сопоставления происходит в самом начале. Вызов `fscanf` присваивает три входных элемента, поэтому описание сигнала выводится только после того, как `n` равно 3. Если `n` не равно `EOF`, то это свидетельствует об ошибке сопоставления, вследствие чего мы выводим в `stderr` подходящее предупреждение, присваиваем переменной `status` значение `EXIT_FAILURE` и выходим из цикла. В качестве последнего возможного варианта `fscanf` возвращает `EOF`, сигнализируя о том, что был достигнут конец файла; в этом случае мы просто выходим из цикла, не изменяя `status`.

Чтобы функция `fscanf` знала, как присваивать входной текст каждому аргументу, ей нужно передать *строку форматирования*. В данном случае она имеет вид `"%d%9s%*[ \t]%99[^\n]"` и состоит из четырех *спецификаторов преобразования*, которые определяют, как ввод, прочитанный из потока, преобразуется в значения объектов, выступающих аргументами строки форматирования. Каждый спецификатор преобразования начинается с символа `%`, за которым могут по очереди идти следующие элементы:

- необязательный символ `*`, который отбрасывает ввод, не присваивая его аргументу;

- необязательное целое число больше нуля, которое определяет максимальную ширину поля (в символах);
- необязательный модификатор длины, определяющий размер объекта;
- символ, обозначающий тип преобразования, которое нужно применить (спецификатор преобразования).

Первый спецификатор преобразования в нашей строке форматирования — `%d`. Он находит первое десятичное целое число (возможно, со знаком), которое должно относиться к номеру сигнала в файле, и сохраняет его в третий аргумент, `sigrec.signum`. Если не указать дополнительный модификатор длины, то размер ввода зависит от типа, который по умолчанию использует спецификатор преобразования. В случае с `d` аргумент должен указывать на `signed int`.

Второй спецификатор преобразования в этой строке форматирования, `%9s`, находит во входном потоке следующую последовательность из непробельных символов, которая относится к названию сигнала, и сохраняет ее в виде строки в четвертый аргумент, `sigrec.signame`. Модификатор длины не дает ввести больше девяти символов и записывает в конец `sigrec.signame` нулевой байт. Если бы в данном примере использовался спецификатор преобразования `%10s`, то мы могли бы столкнуться с переполнением буфера. Но спецификатор `%9s` тоже может не справиться с чтением всей строки, и в этом случае произойдет ошибка сопоставления. При чтении данных в буфер фиксированного размера, как это делаем мы, желательно проверять ввод, длина которого совпадает с длиной буфера или немного ее превышает, чтобы избежать переполнения и убедиться в том, что строка заканчивается нуль-символом.

Пока пропустим третий спецификатор преобразования и сразу перейдем к четвертому, `%99[^\n]`. Он выглядит довольно причудливо и соответствует полю с описанием сигнала. В квадратных скобках (`[]`) находится *множество поиска*, похожее на регулярное выражение. В этом множестве используется знак циркумфлекс (`^`) для исключения символов `\n`. Таким образом, `%99[^\n]` читает описание, пока не дойдет до `\n` (или `EOF`), и сохраняет его в пятый аргумент, `sigrec.sigdesc`. Программисты на C часто задействуют этот синтаксис для чтения строчек целиком. Данный спецификатор преобразования также содержит максимальную длину строки, 99, чтобы не допустить переполнения буфера.

Наконец, вернемся к третьему спецификатору преобразования, `%*[ \t]`. Как мы только что видели, четвертый спецификатор считывает все символы после ID сигнала. К сожалению, это относится и к пробелу между ID и началом описания. Спецификатор `%*[ \t]` должен захватить любые пробелы или символы горизонтальной табуляции между этими двумя полями и предотвратить их присваивание с помощью знака `*`. В данное множество поиска можно включить и другие пробельные символы.

Наконец, мы закрываем файл, вызывая функцию `fclose` ❹.

Чтение из двоичных потоков и запись в них

Функции `fread` и `fwrite` из стандартной библиотеки C работают с двоичными потоками. Сигнатура функции `fwrite` выглядит так:

```
size_t fwrite(const void * restrict ptr, size_t size, size_t nmemb,
  FILE * restrict stream);
```

Эта функция записывает в `stream` до `nmemb` элементов размером `size` байт из массива, на который указывает `ptr`. Для этого она преобразует каждый объект в массив типа `unsigned char` (любой объект может быть приведен к массиву данного типа) и вызывает функцию `fputc`, чтобы по порядку записать значение каждого символа массива. Текущая позиция файла в потоке сдвигается вперед на количество успешно записанных символов.

В листинге 8.6 показано, как с помощью функции `fwrite` записать сведения о сигналах в файл `signals.txt`.

Листинг 8.6. Запись в двоичный файл с использованием прямого ввода/вывода

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

typedef struct sigrecord {
  int signum;
  char signame[10];
  char sigdesc[100];
} sigrecord;

int main(void) {
  int status = EXIT_SUCCESS;
```

```
  FILE *fp;
  sigrecord sigrec;

❶ if ((fp = fopen("signals.txt", "wb")) == NULL) {
    fputs("Cannot open signals.txt file\n", stderr);
    return EXIT_FAILURE;
  }

❷ sigrecord sigrec30 = { 30, "USR1", "user-defined signal 1" };
  sigrecord sigrec31 = {
    .signum = 31, .signame = "USR2", .sigdesc = "user-defined signal 2"
  };

  size_t size = sizeof(sigrecord);

❸ if (fwrite(&sigrec30, size, 1, fp) != 1) {
    fputs("Cannot write sigrec30 to signals.txt file\n", stderr);
    status = EXIT_FAILURE;
    goto close_files;
  }

  if (fwrite(&sigrec31, size, 1, fp) != 1) {
    fputs("Cannot write sigrec31 to signals.txt file\n", stderr);
    status = EXIT_FAILURE;
  }

close_files:
  if (fclose(fp) == EOF) {
    fputs("Failed to close file\n", stderr);
    status = EXIT_FAILURE;
  }

  return status;
}
```

Мы открываем файл `signals.txt` в режиме `wb` ❶, чтобы создать двоичный поток. Объявляем две структуры `sigrecord` ❷ и инициализируем их с помощью значений сигналов, которые нам нужно сохранить в файл. Второй структуре, `sigrec31`, присваиваются назначенные инициализаторы для демонстрации. Оба стиля инициализации ведут себя одинаково; назначенные инициализаторы делают объявление более ясным, хоть и не таким компактным. Процесс записи начинается в строчке ❸. Мы проверяем значение, возвращаемое каждым вызовом функции `fwrite`, и убеждаемся в том, что он записал нужное нам количество элементов.

В листинге 8.7 используется функция `fread` для чтения данных, которые мы только что записали в файл `signals.txt`.

Листинг 8.7. Чтение из двоичного файла с помощью прямого ввода/вывода

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

typedef struct sigrecord {
  int signum;
  char signame[10];
  char sigdesc[100];
} sigrecord;

int main(void) {
  int status = EXIT_SUCCESS;
  FILE *fp;
  sigrecord sigrec;
  size_t size = sizeof(sigrecord);

❶ if ((fp = fopen("signals.txt", "rb")) == NULL) {
    fputs("Cannot open signals.txt file\n", stderr);
    return EXIT_FAILURE;
  }

  // читаем второй сигнал
❷ if (fseek(fp, size, SEEK_SET) != 0) {
    fputs("fseek in signals.txt file failed\n", stderr);
    status = EXIT_FAILURE;
    goto close_files;
  }

❸ if (fread(&sigrec, size, 1, fp) != 1) {
    fputs("Cannot read from signals.txt file\n", stderr);
    status = EXIT_FAILURE;
    goto close_files;
  }

  printf(
    "Signal\n number = %d\n name = %s\n description = %s\n\n",
    sigrec.signum, sigrec.signame, sigrec.sigdesc
  );

close_files:
  fclose(fp);
  return status;
}
```

Мы открываем этот двоичный файл в режиме для чтения, `rb` ❶. Затем этот пример становится немного интереснее: программа считывает и выводит не весь файл целиком, а лишь информацию об определенном

сигнале. Чтобы указать, какой сигнал следует прочитать, программе можно передать аргумент, но в этом примере мы вручную прописали в коде второй сигнал. Программа вызывает функцию `fseek` ❷, чтобы установить текущую позицию файла для потока, на который указывает `fp`. Ранее в этой главе уже упоминалось, что текущая позиция определяет, с каким участком файла будет работать следующая операция ввода/вывода. В случае с двоичным потоком новая позиция устанавливается путем добавления смещения (измеряемого в байтах) к позиции, указанной в последнем аргументе (в данном случае это `SEEK_SET` — начало файла). Первый сигнал в файле имеет позицию 0, а позиция каждого следующего сигнала является произведением его порядкового номера на размер структуры.

После того как текущая позиция файла будет установлена в начало второго сигнала, мы вызываем функцию `fread` ❸, чтобы прочитать данные из двоичного файла в структуру, на которую указывает `&sigrec`. По аналогии с `fwrite` функция `fread` читает из потока, на который указывает `fp`, не больше одного элемента размером `size`. В большинстве случаев этот объект имеет те же размер и тип, что и у вызова `fwrite`. Текущая позиция файла для потока сдвигается на количество успешно прочитанных символов. Мы проверяем возвращаемое из `fread` значение и убеждаемся в том, что данный вызов прочитал нужное нам количество элементов.

Двоичные файлы могут иметь разные форматы. В частности, порядок следования байтов в двоичном представлении числа может зависеть от системы. Байты могут быть упорядочены *от старшего к младшему (big-endian ordering)* и *от младшего к старшему (little-endian ordering)*. Возьмем, к примеру, беззнаковое шестнадцатеричное число `0x1234`, для представления которого нужно по меньшей мере два байта. В формате от старшего к младшему этими двумя байтами будут `0x12` и `0x34`, а в формате от младшего к старшему — `0x34` и `0x12`. В процессорах Intel и AMD байты следуют от младшего к старшему, а процессоры семейств ARM и POWER могут переключаться между этими форматами. Однако в сетевых протоколах, таких как IP, TCP и UDP, доминирует порядок от старшего к младшему. Когда двоичный файл, созданный на одном компьютере, считывается на другом, может возникнуть проблема, если эти компьютеры используют разные форматы. Чтобы этого избежать, всегда сохраняйте двоичные данные с каким-то одним порядком следования байтов или добавьте в файл поле с обозначением выбранного формата.

Резюме

В этой главе вы познакомились с вводом/выводом и с различными аспектами потоков из стандарта C, такими как буферизация, стандартные потоки, ориентация и различия между текстовыми и двоичными потоками.

Затем вы научились создавать, открывать и закрывать файлы с помощью стандартной библиотеки C и API POSIX. Вы узнали, как читать и записывать символы, строки и форматированный текст и как происходит ввод/вывод в двоичных потоках. Мы обсудили сброс потоков на диск, установку текущей позиции в файле и удаление/переименование файлов. Наконец, рассмотрели временные файлы и увидели, почему их не следует использовать.

В следующей главе речь пойдет о процессе компиляции и возможностях препроцессора, таких как подключение файлов, условная компиляция и макросы.

9 Препроцессор (в соавторстве с Аароном Баллманом)

Препроцессор — это часть компилятора C, которая выполняется на раннем этапе компиляции и преобразует исходный код перед его трансляцией, например вставляя код из одного файла (обычно заголовочного) в другой (обычно исходный). В ходе разворачивания макроса препроцессор может также автоматически заменить заданный идентификатор каким-то участком исходного кода. В этой главе вы научитесь с его помощью подключать файлы, определять объектные и функциональные макросы и условно компилировать код в зависимости от свойств реализации.

Процесс компиляции

Сначала посмотрим, какое место препроцессор занимает в процессе компиляции. На концептуальном уровне компиляция представляет собой конвейер из восьми этапов (рис. 9.1). Каждый из них подготавливает код для прохождения следующего этапа.

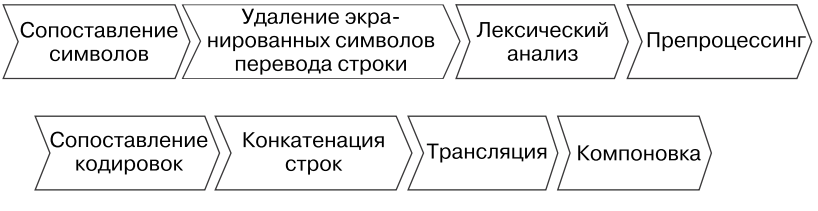

Рис. 9.1. Этапы компиляции

Препроцессор выполняется перед тем, как исходный код транслируется в объектный. Это позволяет ему модифицировать текст, написанный пользователем, *до* его обработки транслятором. В связи с этим препроцессор располагает ограниченным количеством семантической информации о компилируемой программе. Ему ничего не известно о функциях, переменных или типах. Он знаком лишь с простейшими элементами синтаксиса, такими как названия заголовочных файлов, идентификаторы, литералы и знаки препинания наподобие `+`, `-` и `!`. Это так называемые *лексемы (tokens)* — самые мелкие составные части компьютерной программы, которые воспринимаются компилятором.

Препроцессор обрабатывает *директивы препроцессора*, которые программист добавляет в исходный код, чтобы изменить его поведение. Перед именем директивы указывается символ `#` — например, `#include`, `#define` или `#if`. Между началом строчки и `#`, а также между `#` и именем директивы могут находиться отступы в виде пробельных символов. Каждая директива препроцессора должна заканчиваться переводом строки.

Директивы заставляют препроцессор выполнить некое действие, которое может повлиять на результаты трансляции. Это значит, транслятор зачастую получает не совсем тот код, который вы написали. Разные компиляторы обычно дают возможность просматривать вывод препроцессора, так называемую *единицу трансляции*, которая передается транслятору. Вам не обязательно знать, что выводит препроцессор, но может быть полезно взглянуть на тот код, который на самом деле получает транслятор. В табл. 9.1 перечислены флаги, с помощью которых популярные компиляторы выводят единицы трансляции. Файлы с выводом препроцессора обычно имеют расширение `.i`.

Таблица 9.1. Вывод единицы трансляции

Компилятор	Пример командной строки
Clang	`clang other-options -E -o output_file.i source.c`
GCC	`gcc other-options -E -o output_file.i source.c`
Visual C++	`cl other-options /P /Fioutput_file.i source.c`

Подключение файлов

Одной из полезнейших функций препроцессора является возможность вставлять содержимое одного исходного файла в другой с помощью директивы `#include`. Подключаемые файлы называют *заголовочными*, чтобы их можно было отличить от *других* файлов с исходным кодом. Заголовочные файлы обычно содержат объявления, предназначенные для применения в других программах. Это самый распространенный способ разделения внешних объявлений функций и объектов с другими частями программы. Вы уже видели в этой книге много примеров подключения заголовочных файлов с функциями из стандартной библиотеки C.

В табл. 9.2 показана программа, состоящая из заголовочного файла `bar.h` и исходного `foo.c`. В исходнике `foo.c` нет непосредственного объявления `func`, хотя эта функция успешно используется внутри `main`. В ходе предобработки директива `#include`, указанная в `foo.c`, подставляет вместо себя содержимое `bar.h`.

Таблица 9.2. Подключение заголовочного файла

Оригинальные исходники	Итоговая единица трансляции

bar.h
 	
`int func(void);`
	`int func(void);`
	
	`int main(void) {`
	`  return func();`
	`}`

foo.c
 	
`#include "bar.h"`
 	
`int main(void) {`
`  return func();`
`}`	

Препроцессор выполняет директивы `#include` по мере того, как они встречаются в коде. В связи с этим подключению свойственна транзитивность: если исходный файл подключает заголовочный файл, который, в свою очередь, подключает другой заголовочный файл, то вывод препроцессора будет включать содержимое их обоих. Например, если взять заголовочные файлы `baz.h` и `bar.h` и исходный `foo.c`, то результат прохождения последнего через препроцессор будет выглядеть так, как показано в табл. 9.3.

Таблица 9.3. Транзитивное подключение заголовочных файлов

Оригинальные исходники	Итоговая единица трансляции

baz.h
 	
`int other_func(void);`
	`int other_func(void);`
	
	`int func(void);`
	
	`int main(void) {`
	`  return func();`
	`}`

bar.h
 	
`#include "baz.h"`
 	
`int func(void);`	

foo.c
 	
`#include "bar.h"`
 	
`int main(void) {`
`  return func();`
`}`	

При компиляции исходного файла `foo.c` препроцессор подключает заголовочный файл `"bar.h"`. Затем препроцессор находит директиву подключения заголовочного файла `"baz.h"` и вставляет объявление `other_func` в итоговую единицу трансляции.

Строки подключения с кавычками и угловыми скобками

При подключении файла можно использовать либо кавычки (например, `#include "foo.h"`), либо угловые скобки (например, `#include <foo.h>`). Различия между этими двумя вариантами синтаксиса определяются реализацией, но обычно состоят в том, по какому пути ищутся подключенные файлы. Например, Clang и GCC используют для поиска файлов, подключенных с применением:

- угловых скобок, *системный путь*, указанный с помощью флага `-isystem`;
- кавычек, *локальный путь*, указанный с помощью флага `-iquote`.

Если вас интересуют конкретные различия между этими двумя вариантами синтаксиса, то сверьтесь с документацией своего компилятора. Обычно заголовочные файлы стандартных или системных библиотек находятся по стандартному системному пути, а файлы проекта — по локальному.

Условная компиляция

Нередко для поддержки разных реализаций приходится писать разный код. Например, у вас могут быть предусмотрены альтернативные варианты функции для разных целевых архитектур. Одно из решений состоит в создании двух файлов с небольшими отличиями и компиляции того из них, который подходит для конкретной реализации. Но вместо этого решение о том, транслировать ли участки кода, ориентированные на определенную платформу, лучше привязать к определению препроцессора.

Условную компиляцию исходного кода можно выполнять с помощью предикатного условия, используя директивы препроцессора `#if`, `#elif` и `#else`. *Предикатное условие* — это управляющее константное выражение, вычисление которого определяет, по какой ветке программы должен пойти препроцессор. Обычно его применяют в сочетании с операцией препроцессора `defined`, проверяющей, является ли заданный идентификатор именем определенного макроса.

Директивы условной компиляции похожи на операторы `if` и `else`. Когда вычисление предикатного условия дает ненулевое значение, препроцессор обрабатывает только ветку `#if`, а остальные — нет. Когда вычисление предикатного условия дает нуль, на подключение проверяется предикат следующей ветки `#elif` (если таковая имеется). Если ни одно предикатное условие не вернуло ненулевое значение, то обрабатывается ветка `#else` (при наличии таковой). Директива препроцессора `#endif` обозначает конец условно компилируемого кода.

Операция `defined` возвращает `1`, если заданный идентификатор определен в качестве макроса, и `0` в противном случае. Например, директивы препроцессорам показанные в листинге 9.1, условно определяют, содержимое какого заголовочного файла войдет в состав единицы трансляции. Предобработанный вывод `foo.c` зависит от того, какой из макросов определен: `_WIN32` или `__ANDROID__`. Если не определен ни тот ни другой, то вывод препроцессора будет пустым.

Листинг 9.1. Условная компиляция

```
/* foo.c */
#if defined(_WIN32)
#include <Windows.h>
```

```
#elif defined(__ANDROID__)
#include <android/log.h>
#endif
```

В отличие от ключевых слов `if` и `else` условная компиляция препроцессора не может использовать фигурные скобки для обозначения блока операторов, управляемого предикатом. Вместо этого препроцессор охватывает все лексемы между директивами `#if`, `#elif` или `#else` (которые идут за предикатом) и следующей сбалансированной лексемой `#elif`, `#else` или `#endif`; при этом лексемы в необрабатываемых ветках условной компиляции пропускаются. Директивы условной компиляции могут быть вложенными.

Директиву `#if defined` *идентификатор* можно сократить до `#ifdef` *идентификатор* или `#if defined(`*идентификатор*`)`, что то же самое. Скобки вокруг идентификатора можно не указывать. Точно так же вместо `#if !defined` *идентификатор* можно записать `#ifndef` *идентификатор*. Директивы `#elif defined` *идентификатор* и `#elif !defined` *идентификатор* сокращать нельзя.

Генерация ошибок

Если препроцессор не может выбрать никакие условные ветки, поскольку никакой подходящей резервной логики не предусмотрено, то директиве условной компиляции может понадобиться вернуть ошибку. Взгляните на пример в листинге 9.2, который с помощью условной компиляции выбирает между заголовочным файлом `<threads.h>` из стандартной библиотеки C и файлом `<pthread.h>` библиотеки для работы с POSIX-потоками. Если оба варианта недоступны, то вы должны сообщить программисту, который портирует программу, о необходимости исправить код.

Листинг 9.2. Генерация ошибки компиляции

```
#if __STDC__ && __STDC_NO_THREADS__ != 1
#include <threads.h>
//---snip---
#elif POSIX_THREADS == 200809L
#include <pthread.h>
//---snip---
#else
int compile_error[-1]; // Провоцируем ошибку компиляции
#endif
```

Данный код генерирует диагностическое сообщение, но не описывает саму проблему. На этот случай в C предусмотрена директива препроцессора `#error`, заставляющая реализацию вывести сообщение об ошибке. При желании вслед за ней можно указать одну или несколько лексем препроцессора, которые станут частью итогового сообщения. Таким образом, мы можем заменить дефектное объявление массива из листинга 9.2 директивной `#error`, как показано в листинге 9.3.

Листинг 9.3. Директива #error

```
#if __STDC__ && __STDC_NO_THREADS__ != 1
#include <threads.h>
//---snip---
#elif POSIX_THREADS == 200809L
#include <pthread.h>
//---snip---
#else
#error Neither <threads.h> nor <pthread.h> is available
#endif
```

Если ни один из библиотечных заголовочных файлов для работы с потоками не доступен, то этот код генерирует следующее сообщение об ошибке:

```
Neither <threads.h> nor <pthread.h> is available
```

Использование стражей включения

При написании заголовочных файлов необходимо позаботиться о том, что/бы программист не мог повторно подключить заголовочный файл в той же единице трансляции. Учитывая транзитивность директив `#include`, вы вполне можете случайно подключить один и тот же заголовочный файл несколько раз (и, возможно, даже вызвать бесконечную рекурсию между ними).

Стражи включения (header guards) гарантируют, что заголовочный файл подключается в каждой единице трансляции только раз. Это паттерн проектирования, который условно компилирует содержимое заголовочного файла в зависимости от того, определен ли соответствующий макрос. Если макрос не обнаружен, вы его определяете, чтобы код не был условно скомпилирован при последующей проверке. В программе, представленной в табл. 9.4, `bar.h` использует страж включения (выделен жирным шрифтом), чтобы избежать (случайного) дублирования заголовочных файлов из `foo.c`.

Таблица 9.4. Стражи включения

Оригинальные исходники	Итоговая единица трансляции

bar.h
 	
`#ifndef BAR_H`
`#define BAR_H`
`int func(void) { return 1; }`
`#endif /* BAR_H */`
	`int func(void) { return 1; }`
	
	`int main(void) {`
	`  return func();`
	`}`

foo.c
 	
`#include "bar.h"`
`#include "bar.h" // Повторное подключение`
`// обычно не настолько`
`// очевидое`
 	
`int main(void) {`
`  return func();`
`}`	

При первом подключении `"bar.h"` проверка отсутствия определения `BAR_H` возвращает `true`. Затем мы определяем макрос `BAR_H` с пустым списком подстановки, благодаря чему определение функции `func` компилируется. При втором подключении `"bar.h"` препроцессор не генерирует никаких лексем, поскольку проверка условной компиляции возвращает `false`. Таким образом, в итоговую единицу трансляции попадает лишь одно определение `func`.

Идентификатор, который используется для защиты заголовочного файла, принято составлять из важных частей пути, имени файла и его расширения, разделенных подчеркиваниями и записанных большими буквами. Например, если у вас есть заголовочный файл, который подключается как `#include "foo/bar/baz.h"`, то в качестве идентификатора стража включения можно выбрать `FOO_BAR_BAZ_H`.

Некоторые IDE автоматически генерируют стражей включения. Не используйте для них зарезервированные идентификаторы, так как это может привести к неопределенному поведению. Если идентификатор начинается с подчеркивания, за которым идет большая буква, то он является зарезервированным. Например, `_FOO_H` зарезервирован, и его лучше не применять для пользовательских стражей включения, даже если вы подключаете файл `_foo.h`. Это может вызвать конфликт с макросом, который определен реализацией, и привести к ошибке компиляции или получению некорректного кода.

Определение макросов

Директива препроцессора `#define` определяет макрос. *Макросы* могут использоваться для определения константных значений или функциональных конструкций с обобщенными параметрами. Определение макроса содержит *список подстановки* (возможно, пустой) — код, который внедряется в единицу трансляции при его разворачивании:

```
#define идентификатор список_подстановки
```

Директива препроцессора `#define` заканчивается переводом строки. В следующем примере списком подстановки для `ARRAY_SIZE` является `100`:

```
#define ARRAY_SIZE 100
int array[ARRAY_SIZE];
```

В этом примере вместо идентификатора `ARRAY_SIZE` подставляется `100`. Если не указать список подстановки, то препроцессор просто удалит имя макроса. Обычно определение макроса можно указать в командной строке компилятора; например, в Clang и GCC для этого предусмотрен флаг `-D`, а в Visual C++ — `/D`. В Clang и GCC параметр командной строки `-DARRAY_SIZE=100` говорит о том, что идентификатор макроса `ARRAY_SIZE` нужно заменить на `100`, что дает тот же результат, который мы получили в предыдущем примере с помощью директивы препроцессора `#define`. Если не указать список подстановки макроса в командной строке, то большинство компиляторов сделают это за вас. Например, параметр `-DFOO`, как правило, идентичен `#define FOO 1`.

Макрос действует до тех пор, пока препроцессор не встретит директиву `#undef`, принадлежащую этому макросу, или конец единицы трансляции. В отличие от объявления переменных или функций область видимости макроса не зависит от структуры каких-либо блоков.

Директива `#define` позволяет определять как объектные, так и функциональные макросы. *Функциональный макрос* является параметризованным, и при вызове ему необходимо передать (возможно, пустой) набор аргументов по аналогии с тем, как вызывается функция. Но в отличие от функций макросы позволяют выполнять операции с символами исходного файла. Это значит, вы можете создать новое имя переменной или ссылаться на исходный файл и строчку, где находится макрос. *Объектные макросы* являются обычными идентификаторами, которые заменяются фрагментами кода.

В табл. 9.5 показана разница между функциональными и объектными макросами. Объектный макрос `FOO` заменяется во время разворачивания лексемами `(1 + 1)`, а вместо функционального макроса `BAR` подставляются лексемы `(1 + (x))`, где `x` — это параметр, указанный при вызове `BAR`.

Таблица 9.5. Определение макросов

Оригинальные исходники	Итоговая единица трансляции

`#define FOO (1 + 1)`
`#define BAR(x) (1 + (x))`
 	
`int i = FOO;`
`int j = BAR(10);`
`int k = BAR(2 + 2);`
	`int i = (1 + 1);`
	`int j = (1 + (10));`
	`int k = (1 + (2 + 2));`

Открывающие скобки в определении функционального макроса должны идти сразу за его именем, без пробела. Если между именем и открывающей скобкой есть пробел, то скобка становится частью списка подстановки, как в случае с объектным макросом `FOO`. Список подстановки заканчивается первым символом перевода строки в определении макроса. Но вы можете сделать макрос многострочным и более понятным, указав перед переводом строки обратную косую черту (`\`). Например, представленное ниже определение макроса обобщенного типа `cbrt`, которое вычисляет кубический корень своего аргумента с плавающей запятой:

```
#define cbrt(X) _Generic((X), \
  long double: cbrtl(X), \
  default: cbrt(X), \
  float: cbrtf(X) \
)
```

более понятно, чем этот код (хотя они эквивалентны):

```
#define cbrt(X) _Generic((X), long double: cbrtl(X), default: cbrt(X),
float: cbrtf(X))
```

Одна из опасностей, подстерегающих при определении макроса, связана с тем, что после объявления макроса любое использование его идентификатора приводит к разворачиванию макроса. Например, в результате разворачивания макроса следующая дефектная программа не скомпилируется:

```
#define foo (1 + 1)
void foo(int i);
```

Дело в том, что определение `foo`, которое транслятор получает от препроцессора, является некорректным и выглядит так:

```
void (1 + 1)(int i);
```

Для того чтобы избежать данной проблемы, во всей программе необходимо соблюдать один и тот же принцип — например, имена макросов должны быть в верхнем регистре или соответствовать неким мнемоническим правилам, как это делается в некоторых стилях венгерской нотации[1].

Переопределить уже определенный макрос можно только путем использования директивы `#undef`. После этого идентификатор больше не представляет макрос. Например, программа, показанная в табл. 9.6, определяет функциональный макрос, подключает заголовочный файл, который его использует, и затем удаляет определение макроса, чтобы его можно было определить позже.

Таблица 9.6. Удаление определений макросов

Оригинальные исходники	Итоговая единица трансляции

header.h
 	
`NAME(first)`
`NAME(second)`
`NAME(third)`
	`enum Names {`
	`  first,`
	`  second,`
	`  third,`
	`};`
	
	`void func(enum Names Name) {`
	`  switch (Name) {`
	`  case first:`
	`  case second:`
	`  case third:`
	`  }`
	`}`

file.c
 	
`enum Names {`
`  #define NAME(X) X,`
`  #include "header.h"`
`  #undef NAME`
`};`
 	
`void func(enum Names Name) {`
`  switch (Name) {`
`  #define NAME(X) case X:`
`  #include "header.h"`
`  #undef NAME`
`  }`
`}`	

[1] Венгерская нотация — это соглашение об именовании идентификаторов, в соответствии с которым имена переменных или функций должны передавать их назначение, разновидность или, в некоторых диалектах, их тип.

При первом использовании макрос NAME объявляет имена перечислителей внутри перечисления Names. После этого макрос NAME становится неопределенным и переопределяется заново, чтобы сгенерировать метки case в операторе switch.

Определения макросов удаляют перед их переопределением, как показано в листинге 9.4.

Листинг 9.4. Распространенный способ безопасного определения макросов

```
#undef NAME
#define NAME(X) X
```

Даже если макрос с таким именем не существует, удаление его определения является безопасным. Этот код работает независимо от того, определен макрос NAME или нет.

Макроподстановка

Функциональные макросы с виду похожи на функции, но ведут себя иначе. Например, позволяют выполнять операции с символами исходного файла. С их помощью можно создавать новые имена переменных или ссылаться на файл и строчку, где находится макрос, чего нельзя добиться при использовании функций. Когда препроцессор обнаруживает идентификатор макроса, он его разворачивает, подставляя вместо него список подстановки, указанный в определении макроса (если таковой имеется).

Препроцессор подставляет вместо любых параметров в списке подстановки функционального макроса соответствующие аргументы, указанные при его вызове, предварительно их развернув. Если перед параметром в списке подстановки находится лексема #, то он заменяется строковым литералом препроцессора, содержащим текст аргумента (этот процесс иногда называют *преобразованием в строку (stringizing)*). Макрос SRINGIZE, показанный в табл. 9.7, преобразует в строку значение x.

Таблица 9.7. Преобразование в строку

Оригинальные исходники	Итоговая единица трансляции
`#define STRINGIZE(x) #x` `const char *str = STRINGIZE(12);`	`const char *str = "12";`

Препроцессор также удаляет все вхождения лексемы ## в список подстановки, объединяя предыдущую лексему со следующей; это называют

вставкой лексемы (или конкатенацией). Макрос `PASTE` в табл. 9.8 создает новый идентификатор, соединяя `foo`, символ подчеркивания (`_`) и `bar`.

Таблица 9.8. Вставка лексемы

Оригинальные исходники	Итоговая единица трансляции
`#define PASTE(x, y) x ## _ ## y` `int PASTE(foo, bar) = 12;`	`int foo_bar = 12;`

После разворачивания макроса препроцессор заново анализирует список подстановки, чтобы развернуть в нем любые вложенные макросы. Если во время этого процесса (в том числе и при анализе развернутых вложенных макросов внутри списка подстановки) обнаруживается имя макроса, то оно не будет развернуто повторно. Более того, если при разворачивании макроса получается фрагмент программного текста, который идентичен директиве препроцессора, то данный фрагмент не будет интерпретирован как настоящая директива.

Если при разворачивании макроса в списке подстановки встречается несколько параметров с одним и тем же именем, то все они будут заменены аргументом, указанным в вызове. Это может иметь неожиданные последствия, если аргумент макроса имеет побочные эффекты, как показано в табл. 9.9. Данная проблема подробно объясняется в правиле PRE31-C стандарта CERT C (избегайте побочных эффектов в аргументах небезопасных макросов).

Таблица 9.9. Разворачивание небезопасного макроса

Оригинальные исходники	Итоговая единица трансляции

`#define bad_abs(x) (x >= 0 ? x : -x)`
 	
`int func(int i) {`
`  return bad_abs(i++);`
`}`
	`int func(int i) {`
	`  return (i++ >= 0 ? i++ : -i++);`
	`}`

В определении макроса, показанном в табл. 9.9, каждое вхождение параметра `x` заменяется вызовом аргумента `i++`, что приводит к двойному инкременту `i`, который программист или тот, кто проверяет исходный код, вполне может не заметить. Параметры наподобие `x`, равно как и сам список подстановки, обычно следует помещать в скобки, как в `((x) >= 0 ? (x) : -(x))`, чтобы между отдельными вхождениями аргумента `x` и элементами списка подстановки не возникло непредвиденной связи.

Еще одна потенциальная неожиданность связана с тем, что запятая при вызове функционального макроса всегда интерпретируется как разделитель его аргументов. Опасность этого показана в табл. 9.10 на примере макроса `ATOMIC_VAR_INIT`, который входит в стандарт C и позволяет инициализировать произвольную атомарную переменную. Данный код не удается транслировать, поскольку запятая в `ATOMIC_VAR_INIT({1, 2})` воспринимается как разделитель аргументов функционального макроса, в результате чего препроцессор считает, что у макроса два синтаксически некорректных аргумента, `{1` и `2}`, вместо одного корректного, `{1, 2}`[1].

Таблица 9.10. Макрос ATOMIC_VAR_INIT

Оригинальные исходники	Итоговая единица трансляции
stdatomic.h `#define ATOMIC_VAR_INIT(value) (value)`	`<error>`

foo.c
 	
`#include <stdatomic.h>`
 	
`struct S {`
`  int x, y;`
`};`
 	
`_Atomic struct S val = ATOMIC_VAR_INIT({1, 2});`	

Макросы с обобщенными типами

Язык программирования C не позволяет перегружать функции в зависимости от типов передаваемых им параметров, как это можно делать в других языках наподобие Java и C++. Но иногда возникает необходимость изменить логику алгоритма с учетом типов его аргументов. Например, в `<math.h>` есть три функции `sin` (`sin`, `sinf` и `sinl`), поскольку каждый из трех типов с плавающей запятой (`double`, `float` и `long double` соответственно) имеет свою точность. Обобщенные выражения выбора позволяют определить единый идентификатор функции, который делегирует выполнение подходящей внутренней реализации в зависимости от типа аргумента, указанного при вызове.

[1] Эта проблема является одной из причин, почему в C17 макрос ATOMIC_VAR_INIT признали устаревшим.

Обобщенное выражение выбора (generic selection expression) привязывает тип своего невычисленного операнда к соответствующему выражению. Если ни один из типов не подходит, то оно может использовать выражение по умолчанию. Применение *макросов с обобщенными типами* (которые содержат обобщенные выражения выбора) позволяет сделать код более удобочитаемым. В табл. 9.11 мы определили такой макрос, чтобы выбрать подходящий вариант функции `sin` из `<math.h>`.

Таблица 9.11. Обобщенное выражение выбора в виде макроса

Оригинальные исходники	_Generic после разворачивания

`#define sin(X) _Generic((X), \`
`  float: sinf, \`
`  double: sin, \`
`  long double: sinl \`
`)(X)`
 	
`int main(void) {`
`  float f = sin(1.5708f);`
`  double d = sin(3.14159);`
`}`
	`int main(void) {`
	`  float f = sinf(1.5708f);`
	`  double d = sin(3.14159);`
	`}`

Управляющее выражение `(X)` в обобщенном выражении выбора не вычисляется; в зависимости от его типа из списка соответствий `type : expr` выбирается функция. Обобщенное выражение выбора останавливается на одном из этих обозначений (`sinf`, `sin` или `sinl`) и выполняет его. В данном примере при первом вызове `sin` аргумент имеет тип `float`, поэтому обобщенный выбор возвращает `sinf`. Во втором случае мы получаем `sin`, поскольку в качестве типа аргумента указан `double`. Поскольку у этого обобщенного выражения выбора нет привязки `default`, в случае, когда `(X)` не соответствует ни одному из предусмотренных типов, возникает ошибка. Если добавить привязку по умолчанию, то она будет соответствовать любым типам, которые еще не были указаны; часть из них могут оказаться неожиданными, такими как указатели или структуры данных.

Использование обобщенных макросов может быть непростой задачей, если тип итогового значения зависит от типа аргумента макроса, как в случае с `sin` в табл. 9.11. Например, было бы ошибкой присвоить результат вызова макроса `sin` объекту определенного типа или передать его в качестве аргумента функции `printf`, поскольку тип объекта или спецификатор формата зависит от того, какая функция будет вызвана: `sin`, `sinf` или `sinl`.

Примеры макросов с обобщенными типами для математических функций можно найти в заголовочном файле `<tgmath.h>` стандартной библиотеки C.

Предустановленные макросы

Некоторые макросы определяются автоматически на уровне реализации и не требуют подключения заголовочных файлов. Они называются *предустановленными*, поскольку их автоматически определяет препроцессор, а не программист. Например, в стандартной библиотеке C есть множество макросов для получения сведений о среде компиляции и предоставления базовых возможностей. Некоторые другие части реализации (такие как компилятор и система, для которой происходит компиляция) тоже автоматически определяют макросы. В табл. 9.12 перечислены некоторые распространенные макросы, входящие в стандарт языка C. Компиляторы Clang и GCC позволяют получить полный список предустановленных макросов с помощью флагов `-E -dM`. Подробности ищите в документации своей платформы.

Таблица 9.12. Предустановленные макросы

Имя макроса	Замена и назначение
`__DATE__`	Строковый литерал даты создания единицы трансляции в формате *Mmm dd yyyy*
`__TIME__`	Строковый литерал времени создания единицы трансляции в формате *hh:mm:ss*
`__FILE__`	Строковый литерал, представляющий предположительное имя текущего исходного файла
`__LINE__`	Целочисленная константа, представляющая предположительный номер текущей исходной строки
`__STDC__`	Целочисленная константа 1, если реализация соответствует стандарту C
`__STDC_HOSTED__`	Целочисленная константа 1, если реализация является полноценной и 0, если минимальной. Этот макрос условно определяется реализацией (необязательно предопределен)
`__STDC_VERSION__`	Целочисленная константа, представляющая версию стандарта C, на которую рассчитан компилятор (например, 201710L в случае с C17)
`__STDC_ISO_10646__`	Целочисленная константа вида *yyyymmL*. Этот макрос условно определяется реализацией. Если данный символ определен, то каждый символ в кодировке Unicode, хранящийся в объекте типа wchar_t, имеет то же значение, что и короткий идентификатор этого символа

Имя макроса	Замена и назначение
`__STDC_UTF_16__`	Целочисленная константа 1, если значения типа char16_t имеют кодировку UTF-16. Этот макрос условно определяется реализацией
`__STDC_UTF_32__`	Целочисленная константа 1, если значения типа char32_t имеют кодировку UTF-32. Этот макрос условно определяется реализацией
`__STDC_NO_ATOMICS__`	Целочисленная константа 1, если реализация не поддерживает атомарные типы, включая квалификатор типа _Atomic, и заголовочный файл <stdatomic.h>. Этот макрос условно определяется реализацией
`__STDC_NO_COMPLEX__`	Целочисленная константа 1, если реализация не поддерживает комплексные типы или заголовочный файл <complex.h>. Этот макрос условно определяется реализацией
`__STDC_NO_THREADS__`	Целочисленная константа 1, если реализация не поддерживает заголовочный файл <threads.h>. Этот макрос условно определяется реализацией
`__STDC_NO_VLA__`	Целочисленная константа 1, если реализация не поддерживает массивы переменной длины или структуры с массивами произвольной длины. Этот макрос условно определяется реализацией

Резюме

В данной главе вы познакомились с некоторыми возможностями препроцессора. Вы научились подключать фрагменты программного текста к единице трансляции, компилировать код в зависимости от условия и генерировать диагностические сообщения, когда это необходимо. Затем вы узнали, как определяются и вызываются макросы, как удаляются их определения и какие макросы определены реализацией.

В следующей главе мы поговорим о том, как разделить программу на несколько единиц трансляции, чтобы ее было удобнее поддерживать.

10 Структура программы (в соавторстве с Аароном Баллманом)

Любая реальная система состоит из разных компонентов, таких как исходные файлы, заголовочные файлы и библиотеки. Многие системы содержат ресурсы, включая изображения, звуки и конфигурационные файлы. Разделение программы на логические компоненты меньшего размера является рекомендуемым подходом к проектированию программного обеспечения, поскольку эти компоненты легче обслуживать по сравнению с одним большим файлом. В этой главе вы научитесь составлять программы из разных модулей, состоящих из исходных и подключаемых файлов. Кроме того, узнаете, как компоновать несколько объектных файлов для создания библиотек и исполняемых файлов.

Принципы компонентного представления

Ничто не мешает вам написать всю программу целиком внутри функции `main` одного исходного файла. Но по мере увеличения размера данной функции за ней становится сложно уследить. В связи с этим программу имеет смысл разбить на компоненты, которые обмениваются информацией через общую границу *(интерфейс)*. В исходном коде, состоящем из компонентов, легче разобраться и его можно использовать сразу в нескольких местах программы или даже в других проектах.

Обычно понимание того, как лучше всего разделить программу, требует опыта. Многие свои решения программисты принимают с оглядкой на производительность. Например, у вас может возникнуть необходимость

в минимизации взаимодействия по интерфейсу с высокой латентностью. Или вы можете возложить проверку полей ввода на клиентский код, чтобы данные не нужно было отправлять на сервер и обратно. В этом разделе мы рассмотрим некоторые принципы проектирования программного обеспечения на основе компонентов.

Связность и зацепление

Помимо производительности, хорошо структурированная программа должна иметь и другие полезные свойства, такие как слабое зацепление и высокая связность. *Связность (cohesion)* — это показатель того, что у элементов программного интерфейса общая цель. Представьте, к примеру, что заголовочный файл предоставляет функции для определения длины строки, вычисления тангенса заданного входного значения и создания потока выполнения. Такой заголовочный файл имеет низкую связность, поскольку доступные в нем функции не имеют никакого отношения друг к другу. А вот заголовочный файл с функциями для определения длины, соединения двух строк в одну и поиска подстроки имеет высокую связность, поскольку вся его функциональность имеет одну и ту же направленность. Следовательно, если вам нужно работать со строками, то достаточно подключить один подходящий заголовочный файл. Точно так же взаимосвязанные определения функций и типов, которые составляют публичный интерфейс, должны предоставляться одним и тем же заголовочным файлом, чтобы сделать этот интерфейс сильно связным и ограниченным по функциональности. Мы обсудим публичные интерфейсы чуть позже в подразделе «Абстракция данных» на с. 253.

Зацепление (coupling) определяет, насколько созависимы программные интерфейсы. Например, заголовочный файл с жестким зацеплением нельзя подключить к программе сам по себе; он должен подключаться вместе с другими заголовочными файлами и в определенном порядке. Интерфейсы могут быть зацепленными по целому ряду причин, таких как совместное использование определенных структур данных, взаимная зависимость между функциями или работа с разделяемым глобальным состоянием. Однако сильное зацепление интерфейсов затрудняет модификацию логики программы, поскольку изменения могут распространяться по всей системе. Всегда старайтесь делать компоненты слабо зацепленными, независимо от того, принадлежат они к публичному интерфейсу или являются подробностями реализации программы.

Разделяя логику программы на отдельные сильно зацепленные компоненты, вы упрощаете ее анализ и тестирование (так как корректность каждого компонента можно проверять независимо от других). Благодаря этому система содержит меньше дефектов и ее становится легче поддерживать.

Повторное использование кода

Повторное использование кода — это методика, согласно которой функциональность реализуется только раз и затем применяется на разных участках программы без дублирования. Дублирование кода может приводить к тонким и неожиданным расхождениям в поведении системы, чрезмерному увеличению размера исполняемых файлов и повышению расходов на сопровождение кода. Как бы то ни было, зачем писать один и тот же код несколько раз?

Самая низкоуровневая единица функциональности, пригодная для повторного использования, — *функция*. В нее можно инкапсулировать любую логику, которая должна повторяться. Если функциональность имеет несущественные вариации, то функцию зачастую можно параметризировать, чтобы она имела многоцелевое назначение. Каждая функция должна выполнять работу, которая не дублируется никакой другой функцией. Их можно объединять, чтобы решать все более сложные задачи.

Инкапсуляция повторно используемой логики в функции может упростить сопровождение кода и устранить дефекты. Например, длину строки с нуль-символом в конце можно определить с помощью простого цикла `for`, но ваш код будет легче сопровождать, если задействовать вместо этого функцию `strlen` из стандартной библиотеки C. Другие программисты уже знакомы с данной функцией, поэтому им будет легче понять, что она может делать по сравнению с циклом `for`. Более того, повторное использование существующих возможностей снижает вероятность появления расхождений в поведении кода, по сравнению с написанием новой реализации в каждом отдельном случае; это также позволит вам легко заменить функциональность более производительным алгоритмом или более безопасной реализацией на глобальном уровне.

При проектировании функциональных интерфейсов необходимо поддерживать баланс между их *общими* и *конкретными* аспектами. Интерфейс, рассчитанный на текущие требования, может быть компактным и эффек-

тивным, но если эти требования поменяются, то его будет сложно адаптировать. Общий интерфейс может предусматривать будущие изменения требований, но быть слишком громоздким для текущих нужд.

Абстракция данных

Абстракция данных — любой программный компонент, рассчитанный на повторное использование и соблюдающий четкое разделение между публичным интерфейсом абстракции и подробностями ее реализации. *Публичный интерфейс* любой абстракции данных состоит из определений типов, объявлений функций и определений констант, необходимых пользователю этой абстракции; он размещается в заголовочном файле. Подробности реализации того, как абстракция воплощена в коде, а также приватные вспомогательные функции размещаются в файле с исходным кодом или в отдельном от публичного интерфейса заголовочном файле. Подобное разделение публичного интерфейса и приватной реализации позволяет изменять последнюю, не нарушая работу кода, зависящего от вашего компонента.

Заголовочные файлы обычно содержат объявления функций и определения типов компонента. Например, `<string.h>` из стандартной библиотеки C предоставляет публичный интерфейс для действий, связанных со строками, а в `<threads.h>` находятся служебные функции для работы с потоками выполнения. Такое логическое разделение позволяет добиться слабого зацепления и высокой связности. Этот подход позволяет обращаться только к тем компонентам, которые вам нужны, сокращая время компиляции и снижая вероятность конфликтов имен. Например, если вы хотите воспользоваться функцией `strlen`, то вам не нужно ничего знать об API для работы с потоками.

Еще один вопрос, о котором нужно подумать, состоит в том, нужно ли явным образом подключить заголовочные файлы, необходимые вашему заголовочному файлу, или пусть этим занимаются сами пользователи. С точки зрения абстрагирования данных заголовочные файлы лучше сделать самодостаточными и подключить к ним любые файлы, которые они задействуют. Если этого не сделать, то пользователям абстракции придется выполнять дополнительные действия, что чревато утечкой подробностей ее реализации. Примеры, представленные в данной книге, должны быть лаконичными и потому не всегда следуют этим рекомендациям.

Исходные файлы реализуют функциональность, объявленную заданным заголовочным файлом или программную логику, ориентированную на определенные задачи и применяемую для выполнения необходимых действий. Например, если у вас есть заголовочный файл `network.h`, описывающий публичный интерфейс для сетевого взаимодействия, то у вас может быть исходный файл `network.c` (или `network_win32.c` для Windows и `network_linux.c` для Linux), который реализует его логику.

Подробности реализации можно разделять между двумя исходными файлами, используя общий заголовочный файл, но он должен храниться отдельно от публичного интерфейса, чтобы случайно не раскрыть эти подробности.

Хорошим примером абстракции данных, в которой основная функциональность отделена от реализации или внутренней структуры, является *коллекция*. Она представляет группу элементов данных и поддерживает их добавление, удаление и проверку наличия того или иного элемента. Коллекцию можно реализовать множеством разных способов. Так, ее можно представить в виде одномерного массива, двоичного дерева, ориентированного (возможно, ациклического) графа или другой структуры. Выбор структуры данных может повлиять на производительность алгоритма в зависимости от того, какую информацию вы представляете и сколько места она занимает. Например, двоичное дерево может быть более подходящей абстракцией для крупных объемов данных, по которым нужно быстро искать, тогда как одномерный массив, скорее всего, лучше подойдет для фиксированного количества элементов небольшого размера. Разделение интерфейса коллекции и реализации внутренней структуры данных позволяет изменять реализацию, не внося изменения в код, который зависит от этого интерфейса.

Непрозрачные типы

Максимально эффективной абстракция данных становится при использовании непрозрачных типов, которые скрывают информацию. В языке C *непрозрачными* (или *приватными*) выступают неполные типы данных, такие как предварительные объявления структур. *Неполным* называют тип, который описывает идентификатор, но не содержит информацию, необходимую для определения размера или внутреннего устройства объектов этого типа. Скрытие структур данных, предназначенных сугубо

для внутреннего применения, препятствует написанию кода, который зависит от подробностей потенциально изменчивой реализации. Неполный тип доступен пользователям абстракции данных, тогда как с полностью определенным типом может работать только реализация.

Представьте, что вам нужно реализовать коллекцию с поддержкой ограниченного количества операций, таких как добавление, удаление и поиск элементов. В следующем примере реализован непрозрачный тип `collection_type`, который скрывает подробности реализации типа данных от пользователей библиотеки. Для этого мы создадим два заголовочных файла: внешний, `collection.h`, который подключается пользователем типа, и внутренний, подключаемый только в файлах, которые реализуют функциональность этого типа данных.

Во внешнем заголовочном файле `collection.h` тип данных `collection_type` определяется как экземпляр структуры `collection_type`, которая имеет неполный тип:

```
typedef struct collection_type collection_type;
// объявление функций
extern errno_t create_collection(collection_type **result);
extern void destroy_collection(collection_type *col);
extern errno_t add_to_collection(collection_type *col, const void *data,
  size_t byteCount);
extern errno_t remove_from_collection(collection_type *col,
  const void *data, size_t byteCount);
extern errno_t find_in_collection(const collection_type *col,
  const void *data, size_t byteCount);
//---snip---
```

Идентификатор `collection_type` выступает псевдонимом структуры `collection_type` (неполного типа). Следовательно, функции в публичном интерфейсе должны принимать указатель на данный тип, а не само значение; это связано с ограничениями, которые накладываются на неполные типы в языке C.

Структура `collection_type` полностью определена во внутреннем заголовочном файле, но не видна пользователям абстракции данных:

```
struct node_type {
  void *data;
  size_t size;
  struct node_type *next;
};
```

```
struct collection_type {
  size_t num_elements;
  struct node_type *head;
};
```

В модулях, которые реализуют абстрактный тип данных, подключается как внешнее, так и внутреннее определение, тогда как пользователи данного типа подключают только внешний заголовочный файл `collection.h`. Это позволяет оставить приватной реализацию типа `collection_type`.

Исполняемые файлы

В главе 9 вы узнали, что компилятор — это конвейер, состоящий из этапов трансляции, и что конечным результатом его работы является объектный код. На последнем этапе, который называется *компоновкой* (или «линковкой»), объектный код со всех единиц трансляции в программе собирается в итоговый файл. Данный файл может быть исполняемым (например, `a.out` или `foo.exe`) и доступным для запуска, библиотечным или узкоспециализированной программой, такой как драйвер устройства или прошивка (машинный код, записываемый в энергонезависимую память). Компоновка позволяет разделить код на отдельные исходные файлы, которые можно компилировать по отдельности, что помогает создавать компоненты, пригодные к повторному использованию.

Библиотеки — это исполняемые компоненты, которые не могут запускаться сами по себе. Вместо этого их можно внедрять в исполняемые программы. Вы можете задействовать возможности библиотеки, подключая к своему исходному коду ее заголовочные файлы и вызывая объявленные в ней функции. Примером этого служит стандартная библиотека C — вы подключаете ее заголовочные файлы, но не компилируете напрямую исходный код, который реализует ее функциональность. Реализация поставляется с предварительно собранной версией библиотечного кода. Используя универсальные библиотечные компоненты, написанные кем-то другим, вы можете сосредоточиться на разработке логики, которая относится к вашей конкретной программе. Например, если вы пишете компьютерную игру, то применение готовых библиотек должно позволить вам сосредоточиться на игровой логике и не беспокоиться о таких низкоуровневых деталях, как получение пользовательского ввода, взаимодействие по сети или отрисовка графики. Библиотеки зачастую позволяют программе, собранной одним компилятором, использовать код, собранный другим компилятором.

Библиотеки компонуются с приложением и могут быть либо статическими, либо динамическими. *Статическая библиотека*, также известная как архив, внедряет свой машинный или объектный код непосредственно в итоговый исполняемый файл, что нередко привязывает ее к определенному выпуску программы. Поскольку данное внедрение происходит на этапе компоновки, содержимое статической библиотеки можно дополнительно оптимизировать под ваши нужды. Библиотечный код, который использует ваша программа, попадает в итоговый исполняемый файл, а все остальное можно выбросить.

Динамическая библиотека, которую еще называют разделяемой или динамически разделяемым объектом, представляет собой исполняемый файл без процедур, предназначенных для его запуска. Она может быть упакована вместе с исполняемым файлом программы или установлена отдельно, но когда ваш код вызывает одну из ее функций, она должна быть доступна. Многие современные операционные системы загружают код динамической библиотеки в память всего один раз и разделяют его между всеми приложениями, которым он нужен. При необходимости вы можете менять версии динамической библиотеки после развертывания своего приложения. Разработка библиотечного кода отдельно от программного имеет свои преимущества и риски. Например, программист может исправить дефекты в динамической библиотеке уже после выпуска приложения, не требуя его перекомпиляции. Но это также может позволить злоумышленнику заменить подлинную библиотеку вредоносной, или же пользователь может случайно установить не ту версию. Кроме того, в новый выпуск библиотеки можно внести *ломающие изменения*, приводящие к несовместимости с существующими приложениями, которые ее задействуют. Статические библиотеки могут работать несколько быстрее, поскольку их объектный (двоичный) код уже включен в исполняемый файл. В целом преимущества использования динамических библиотек перевешивают недостатки.

Каждая библиотека содержит по меньшей мере один заголовочный файл, описывающий ее публичный интерфейс, и как минимум один исходный файл, который реализует ее логику. Структурирование кода в виде набора библиотек может быть полезно, даже если он не компилируется в настоящие библиотечные файлы. Применение отдельной библиотеки препятствует случайному проектированию сильно зацепленных интерфейсов, в которых один компонент знает о внутреннем устройстве другого.

Компоновка

Компоновка (линковка) — это процесс, который определяет, является интерфейс публичным или приватным и ссылаются ли какие-либо два идентификатора на одну и ту же сущность. В языке C компоновка бывает внешней и внутренней, или же ее может не быть вообще. Когда объявление имеет *внешнюю компоновку*, все его идентификаторы, на каком бы участке программы ни находились, ссылаются на одно и то же (функцию или объект). При *внутренней компоновке* идентификаторы объявления ссылаются на одну и ту же сущность только в рамках той единицы трансляции, где находится это объявление. Если две единицы трансляции ссылаются на один и тот же идентификатор с внутренней компоновкой, то в конечном счете работают с разными экземплярами сущности. Если у объявления *нет компоновки*, то оно представлено уникальной сущностью в каждой единице трансляции.

Компоновка объявления либо указывается явно, либо подразумевается неявно. Если вы объявляете сущность на уровне файла без использования спецификаторов `extern` и `static`, то ей неявно назначается внешняя компоновка. К сущностям без компоновки относятся параметры функций, идентификаторы, объявляемые на уровне блока без применения спецификатора класса хранения `extern`, а также константы-перечисления.

В листинге 10.1 показаны примеры объявлений с каждым видом компоновки.

Листинг 10.1. Примеры внешней, внутренней и отсутствующей компоновки

```
static int i; // i объявляется с явной внутренней компоновкой
extern void foo(int j) {
  // foo объявляется с явной внешней компоновкой
  // у j нет компоновки, так как это параметр
}
```

Если на уровне файла в объявлении идентификатора явно указать спецификатор класса хранения `static`, то он будет иметь внутреннюю компоновку. Ключевое слово `static` назначает внутреннюю компоновку только сущностям в области видимости файла. Применение его к переменной на уровне блока породит идентификатор без компоновки, зато этой переменной будет назначен статический срок хранения. Как вы помните, статический срок хранения означает, что время жизни сущности охватывает весь

период выполнения программы, а хранимое значение инициализируется всего раз, еще до запуска. Очевидно, что зависимость спецификатора `static` от контекста вносит путаницу и поэтому часто используется на собеседованиях.

Вы можете создать идентификатор с внешней компоновкой, объявив его с помощью спецификатора класса хранения `extern`. Но это работает, только если вы еще не объявляли компоновку для данного идентификатора. Ключевое слово `extern` не оказывает на идентификатор никакого влияния, если ему уже была назначена какая-то компоновка.

Объявления с конфликтующими компоновками могут вызывать неопределенное поведение; более подробную информацию об этом можно получить в правиле DCL36-C стандарта CERT C (не объявляйте идентификатор с конфликтующими классами компоновки).

В табл. 10.1 показаны примеры объявлений с явной и неявной компоновкой.

Таблица 10.1. Примеры явной и неявной компоновки

Косвенная и ручная компоновка

foo.c
 	
`void func(int i) { // Неявная внешняя компоновка`
`  // у i нет компоновки`
`}`
 	
`static void bar(void); // Внутренняя компоновка, bar не из bar.c`
 	
`extern void bar(void) {`
`  // bar по-прежнему имеет внутреннюю компоновку, так как начальное`
`  // объявление было статическим; спецификатор extern в данном случае`
`  // ни на что не влияет`
`}`

`bar.c`
 	
`extern void func(int i); // Явная внешняя компоновка`
 	
`static void bar(void) { // Internal Внутренняя компоновка, bar не из foo.c`
`  func(12); // Вызывает func из foo.c`
`}`
`int i; // Внешняя компоновка; не конфликтует с i из foo.c или bar.c`
`void baz(int k) { // Неявная внешняя компоновка`
`  bar(); // Вызываем bar из bar.c, не foo.c`
`}`

Идентификаторы в вашем публичном интерфейсе должны иметь внешнюю компоновку, чтобы их можно было вызывать из-за пределов их единицы трансляции. В то же время идентификаторы, относящиеся к подробностям реализации, нужно объявлять либо с внутренней компоновкой, либо без таковой. Обычно, чтобы этого достичь, функции публичного интерфейса объявляют в заголовочном файле со спецификатором класса хранения `extern` или без него (внешняя компоновка назначается объявлениям автоматически, но в использовании `extern` нет никакого вреда) и похожим образом размещают их определения в исходном файле.

Однако внутри исходного файла все объявления, которые выступают подробностями реализации, должны содержать спецификатор `static`; это позволяет сделать их приватными и доступными только в данном файле. Вы можете подключить публичный интерфейс, объявленный в заголовочном файле, используя директиву препроцессора `#include`, чтобы обратиться к нему из другого файла. Сущности, которые объявлены на уровне файла и не должны быть видны за его пределами, обычно лучше делать статическими (`static`). Этот подход ограничивает загромождение глобального пространства имен и снижает вероятность непредвиденного взаимодействия между единицами трансляции.

Структурирование простой программы

Чтобы научиться структурировать настоящие сложные программы, начнем с несложного примера, определяющего, является ли число простым. *Простым* называют натуральное число больше 1, которое нельзя получить путем умножения двух меньших натуральных чисел. Мы напишем два отдельных компонента: статическую библиотеку, которая будет заниматься проверкой, и приложение командной строки с пользовательским интерфейсом к ней.

Программа `primetest` принимает на вход список целочисленных значений, разделенных пробелами, и выводит информацию о том, является ли каждое из них простым числом. Если какая-либо часть ввода окажется некорректной, то программа выведет информативное сообщение с описанием того, как пользоваться ее интерфейсом.

Прежде чем приступать к структурированию программы, рассмотрим пользовательский интерфейс. Вначале мы выводим справочную информацию для программы командной строки, как показано в листинге 10.2.

Листинг 10.2. Вывод справочной информации

```
// Выводим справочный текст в командной строке
static void print_help(void) {
  printf("%s", "primetest num1 [num2 num3 ... numN]\n\n");
  printf("%s", "Tests positive integers for primality. Supports testing
");
  printf("%s [2-%llu].\n", "numbers in the range", ULLONG_MAX);
}
```

Функция `print_help` состоит из трех отдельных вызовов `printf`, которые записывают в стандартный вывод текст с объяснением о том, как использовать эту команду.

Аргументы командной строки передаются программе в текстовом виде, поэтому мы определяем служебную функцию, чтобы преобразовать их в целочисленные значения, как показано в листинге 10.3.

Листинг 10.3. Преобразование отдельного аргумента командной строки

```
// Преобразует строковый аргумент arg в значение unsigned long long,
// на которое ссылается val
// Возвращает true, если преобразование аргументов было успешным,
// и false, если нет
static bool convert_arg(const char *arg, unsigned long long *val) {
  char *end;

  // strtoll возвращает внутренний индикатор ошибки; очистите errno
  // перед вызовом
  errno = 0;
  *val = strtoull(arg, &end, 10);

  // Отслеживаем ошибки, когда вызов возвращает контрольное значение
  // и устанавливает errno
  if ((*val == ULLONG_MAX) && errno) return false;
  if (*val == 0 && errno) return false;
  if (end == arg) return false;

  // Если мы попали сюда, нам удалось преобразовать аргумент.
  // Но мы хотим допускать только
  // значения больше 1, поэтомы мы отбрасываем все значения <= 1
  if (*val <= 1) return false;
  return true;
}
```

Функция `convert_arg` принимает на вход строковый аргумент и использует выходной параметр, чтобы вернуть его преобразованную версию

(*выходной параметр* возвращает вызывающей стороне результат работы функции в обход оператора `return`, что позволяет вернуть сразу несколько значений). Функция возвращает `true`, если преобразование прошло успешно, и `false` в противном случае. Чтобы превратить строку в целочисленное значение `unsigned long long`, используется вызов `strtoull` с надлежащей обработкой возможных ошибок. Кроме того, поскольку определение простых чисел исключает 0, 1 и отрицательные значения, `convert_arg` воспринимает их как некорректный ввод.

Как видно в листинге 10.4, мы вызываем `convert_arg` в функции `convert_command_line_args`, которая циклически перебирает все предоставленные аргументы командной строки и пытается преобразовать их в целые числа.

Листинг 10.4. Обработка всех аргументов командной строки

```
static unsigned long long *convert_command_line_args(int argc,
                                                     const char *argv[],
                                                     size_t *num_args) {
  *num_args = 0;

  if (argc <= 1) {
    // Непредоставление никаких аргументов командной
    // строки (первый аргумент — это имя исполняемой программы)
    print_help();
    return NULL;
  }

  // Мы знаем, какое максимальное количество аргументов мог бы
  // передать пользователь, поэтому выделяем массив, способный
  // их вместить. Вычитаем один, имя самой программы. Если выделение
  // оказывается неудачным, считаем это ошибкой преобразования
  // (можно вызвать free(NULL)).
  unsigned long long *args =
     (unsigned long long *)malloc(sizeof(unsigned long long) * (argc - 1));
  bool failed_conversion = (args == NULL);
  for (int i = 1; i < argc && !failed_conversion; ++i) {
    // Пытаемся преобразовать аргумент в целое число. Если его не удается
    // преобразовать, присваиваем failed_conversion значение true
    unsigned long long one_arg;
    failed_conversion |= !convert_arg(argv[i], &one_arg);
    args[i - 1] = one_arg;
  }

  if (failed_conversion) {
    // Освобождаем массивм, выводим справочный текст и выходим
    free(args);
```

```
    print_help();
    return NULL;
  }

  *num_args = argc - 1;
  return args;
}
```

Если какой-либо из аргументов не удается преобразовать, то вызывается функция `print_help`, которая объясняет пользователю, как правильно работать с программой, после чего возвращается нулевой указатель. Данная функция отвечает за выделение буфера, в который должен поместиться массив с целыми числами. Вдобавок она обрабатывает любые ошибки, такие как нехватка памяти или невозможность преобразования аргумента. Если функция `convert_command_line_args` завершается успешно, то возвращает вызывающей стороне целочисленный массив и записывает количество преобразованных аргументов в параметр `num_args`. Память для возвращенного массива была выделена динамически, и после того, как она больше не нужна, ее необходимо освободить.

Определить, является ли число простым, можно несколькими способами. Самый примитивный подход состоит в том, чтобы проверить, делится ли значение N на $[2..N-1]$ без остатка. Производительность данного решения ухудшается по мере увеличения N. Вместо этого мы воспользуемся одним из многочисленных алгоритмов, предназначенных для определения простых чисел. В листинге 10.5 показана вероятностная реализация теста Миллера — Рабина, которая позволяет с определенной вероятностью предположить, является ли число простым (Шуф, 2008). Если вас интересует математическая сторона алгоритма Миллера — Рабина, то можете ознакомиться с публикацией Рене Шуфа.

Листинг 10.5. Алгоритм Миллера — Рабина для проверки простоты чисел

```
static unsigned long long power(unsigned long long x, unsigned long long y,
                                unsigned long long p) {
  unsigned long long result = 1;
  x %= p;

  while (y) {
    if (y & 1) result = (result * x) % p;
    y >>= 1;
    x = (x * x) % p;
  }
```

```
  return result;
}

static bool miller_rabin_test(unsigned long long d, unsigned long long n) {
  unsigned long long a = 2 + rand() % (n - 4);
  unsigned long long x = power(a, d, n);

  if (x == 1 || x == n - 1) return true;

  while (d != n - 1) {
    x = (x * x) % n;
    d *= 2;

    if (x == 1) return false;
    if (x == n - 1) return true;
  }
  return false;
}
```

Интерфейсом для теста Миллера — Рабина служит функция `is_prime`, показанная в листинге 10.6. Она принимает два аргумента: число, которое нужно проверить (`n`), и количество необходимых проверок (`k`). Чем больше значение `k`, тем точнее результат, но хуже производительность. Мы поместим алгоритм из листинга 10.5 в статическую библиотеку вместе с функцией `is_prime`, которая будет выступать ее публичным интерфейсом.

Листинг 10.6. Интерфейс к алгоритму Миллера — Рабина для проверки простоты чисел

```
bool is_prime(unsigned long long n, unsigned int k) {
  if (n <= 1 || n == 4) return false;
  if (n <= 3) return true;

  unsigned long long d = n - 1;
  while (d % 2 == 0) d /= 2;

  for (; k != 0; --k) {
    if (!miller_rabin_test(d, n)) return false;
  }
  return true;
}
```

Наконец, нам нужно объединить эти служебные функции в программу. В листинге 10.7 показана реализация функции `main`. Она выполняет заданное количество проверок Миллера — Рабина и сообщает одно из двух: либо число простое с определенной долей вероятности, либо точно нет.

Функция `main` также занимается освобождением памяти, которую выделила функция `convert_command_line_args`.

Листинг 10.7. Функция main

```
int main(int argc, char *argv[]) {
  size_t num_args;
  unsigned long long *vals = convert_command_line_args(argc, argv, &num_
args);

  if (!vals) return EXIT_FAILURE;

  for (size_t i = 0; i < num_args; ++i) {
    printf("%llu is %s.\n", vals[i],
           is_prime(vals[i], 100) ? "probably prime" : "not prime");
  }
  free(vals);
  return EXIT_SUCCESS;
}
```

Функция `main` вызывает `convert_command_line_args`, чтобы преобразовать аргументы командной строки в целочисленный массив типа `unsigned long long`. Программа циклически перебирает каждый элемент этого массива и вызывает `is_prime`, чтобы определить, является ли он простым (с определенной долей вероятности), используя тест Миллера — Рабина, реализованный функцией `is_prime`.

Теперь, реализовав программную логику, мы создадим необходимые артефакты сборки. Наша цель — получить статическую библиотеку с реализацией теста Миллера — Рабина и приложение командной строки для работы с ней.

Сборка кода

Создайте новый файл `isprime.c`, скопируйте в него код из листингов 10.5 и 10.6 (именно в таком порядке) и добавьте в самом начале директивы `#include` для `"isprime.h"` и `<stdlib.h>`. Кавычки и угловые скобки сообщают препроцессору, где нужно искать соответствующие заголовочные файлы (это обсуждалось в главе 9). Затем создайте заголовочный файл `isprime.h` и скопируйте в него содержимое листинга 10.8; это публичный интерфейс статической библиотеки со стражем включения.

Листинг 10.8. Публичный интерфейс для статической библиотеки

```
#ifndef PRIMETEST_IS_PRIME_H
#define PRIMETEST_IS_PRIME_H

#include <stdbool.h>

bool is_prime(unsigned long long n, unsigned k);

#endif // PRIMETEST_IS_PRIME_H
```

Создайте новый файл `driver.c` с кодом из листингов 10.2, 10.3, 10.4 и 10.7 (именно в таком порядке) и добавьте в самое его начало директивы `#include` для следующих заголовочных файлов: `"isprime.h"`, `<assert.h>`, `<errno.h>`, `<limits.h>`, `<stdbool.h>`, `<stdio.h>` и `<stdlib.h>`. В нашем примере все три файла находятся в одном каталоге, но в реальном проекте они, скорее всего, хранились бы в разных местах в соответствии с тем, как это принято в вашей конкретной системе сборки. Создайте локальный каталог bin, в котором будут созданы артефакты сборки данного примера.

Для создания статической библиотеки и исполняемой программы мы используем Clang, но те же аргументы командной строки поддерживаются и в GCC, поэтому в данном примере подходят оба компилятора. Сначала скомпилируем оба исходника с кодом на C в объектные файлы и поместим их в каталог bin:

```
% clang -c -std=c17 -Wall -Wextra -pedantic -Werror isprime.c -o bin/
isprime.o
% clang -c -std=c17 -Wall -Wextra -pedantic -Werror driver.c -o bin/
driver.o
```

Если при выполнении этой команды возникнет следующая ошибка:

```
unable to open output file 'bin/isprime.o': 'No such file or directory'
```

то создайте локальный каталог bin и попробуйте снова. Флаг `-c` заставляет Clang скомпилировать исходный код в объектный файл, не прибегая к вызову компоновщика, который сгенерировал бы исполняемый результат. Объектные файлы нам понадобятся для создания библиотеки. Флаг `-o` определяет путь к итоговому файлу.

После выполнения команды в каталоге bin должны находиться два файла с объектным кодом для каждой единицы трансляции: `isprime.o` и `driver.o`. Мы могли бы скомпоновать их напрямую, чтобы получить исполняемую программу. Но в данном случае мы хотим сделать статическую библи-

отеку (которую по историческим причинам также называют *архивом*). Выполните команду `ar`, чтобы сгенерировать статическую библиотеку `libPrimalityUtilities.a` в каталоге bin:

```
% ar rcs bin/libPrimalityUtilities.a bin/isprime.o
```

Параметр `r` заставляет команду `ar` заменить любые существующие файлы новыми, параметр `c` создает архив, а параметр `s` записывает в него индекс объектного файла (что эквивалентно выполнению команды `ranlib`). В результате получится единый архивный файл; его структура позволяет извлекать из него отдельные объектные файлы, из которых он составлен, — по аналогии со сжатыми файлами TAR или ZIP. Для статических библиотек в системах Linux принято использовать префикс `lib` и расширение `.a`.

Теперь вы можете скомпоновать объектный файл `driver.o` со статической библиотекой `libPrimalityUtilities.a`, чтобы получить исполняемый файл `primetest`. Это можно сделать либо с помощью компилятора без флага `-c` (в таком случае компилятор вызовет стандартный системный компоновщик с подходящими аргументами), либо вызвав компоновщик напрямую. Вызовем стандартный системный компоновщик, воспользовавшись компилятором:

```
% clang bin/driver.o -Lbin -lPrimalityUtilities -o bin/primetest
```

Флаг `-L` заставляет компоновщик искать компонуемые библиотеки в каталоге bin, а флаг `-l` инициирует компоновку библиотеки `libPrimalityUtilities.a` с результатом. Не указывайте в аргументе командной строки префикс `lib` и расширение `.a`, поскольку компоновщик добавляет их автоматически. Например, чтобы скомпоновать код с математической библиотекой `libm`, в качестве цели компоновки нужно указать `-lm`. Как и при компиляции исходных файлов, название результата компоновки указывается с помощью флага `-o`.

Теперь вы можете с помощью этой программы проверить, что число точно составное или, вероятно, простое (с определенной долей вероятности). Попробуйте передать ей отрицательные, заведомо простые и непростые числа, а также некорректный ввод, как показано в листинге 10.9.

Листинг 10.9. Выполнение программы program с разным вводом

```
% ./bin/primetest 899180
899180 is not prime
% ./bin/primetest 8675309
```

```
8675309 is probably prime
% ./bin/primetest 0
primetest num1 [num2 num3 ... numN]

Tests positive integers for primality. Supports testing numbers
in the range
[2-18446744073709551615].
```

Число 8 675 309 является простым.

Резюме

В этой главе вы узнали о преимуществах слабого зацепления и сильной связности, абстракции данных и повторного использования кода. Вдобавок познакомились с сопутствующими конструкциями языка, такими как непрозрачные типы данных и компоновка. Мы обсудили рекомендуемые методы структурирования кода в проектах и рассмотрели пример сборки демонстрационной программы из разного рода исполняемых компонентов.

В следующей главе речь пойдет об инструментах и методиках создания высококачественных систем, включая утверждения, отладку, тестирование, а также статический и динамический анализ.

11 Отладка, тестирование и анализ

В этой главе описываются инструменты и методики, направленные на создание корректных, эффективных, безопасных, защищенных и надежных программ, включая статические и динамические утверждения (времени компиляции и выполнения соответственно), отладку, тестирование, а также статический и динамический анализ. Кроме того, в данной главе обсуждаются флаги компилятора, рекомендуемые к использованию на разных этапах процесса разработки.

Утверждения

Утверждение позволяет убедиться в том, что предположение, сделанное вами во время реализации вашей программы, остается действительным. Это функция с булевым значением, известным как *предикат*, выражающим логическое суждение о программе. Язык C поддерживает статические утверждения, которые могут быть проверены на этапе компиляции с помощью `static_assert`, и динамические, которые проверяются во время выполнения программы с помощью `assert`. Макросы `assert` и `static_assert` определены в заголовочном файле `<assert.h>`.

Статические утверждения

Статические утверждения можно выразить с помощью макроса `static_assert` следующим образом:

```
static_assert(целочисленное_константное_выражение, строковый_литерал);
```

Если значение целочисленного константного выражения не равно 0, то объявление `static_assert` ничего не делает. В противном случае оно заставит компилятор вывести диагностическое сообщение с текстом строкового литерала, который вы указали.

Статические утверждения позволяют проверять во время компиляции предположения о конкретных аспектах поведения, которые зависят от реализации. Любое изменение этих аспектов будет диагностировано в ходе сборки кода.

Рассмотрим три примера использования статических утверждений. В первом, показанном в листинге 11.1, `static_assert` проверяет, нет ли у структуры `packed` заполняющих байтов.

Листинг 11.1. Проверка отсутствия в структуре заполняющих байтов с помощью static_assert

```
#include <assert.h>

struct packed {
  unsigned int i;
  char *p;
};

static_assert(
  sizeof(struct packed) == sizeof(unsigned int) + sizeof(char *),
  "struct packed must not have any padding"
);
```

В данном примере предикат статического утверждения проверяет, совпадает ли размер структуры `packed` с совокупным размером ее членов `unsigned int` и `char *`. Поскольку макрос `static_assert` — это объявление, он может находиться в области видимости файла, сразу за определением структуры, свойство которой он проверяет.

Функция `clear_stdin`, представленная в листинге 11.2, читает символы из `stdin` с помощью `getchar`, пока не достигнет конца файла. Каждый символ извлекается в виде `unsigned char` и приводится к типу `int`. Чтобы определить, были ли прочитаны все доступные символы, результат работы функции `getchar` обычно сравнивают с `EOF`, зачастую делая это в цикле `do...while`. Чтобы цикл работал правильно, условие его прерывания должно уметь различать символы и `EOF`. Однако стандарт C позволяет использовать для `unsigned char` и `int` один и тот же диапазон, как следствие,

в некоторых реализациях такая проверка на EOF может давать ложные срабатывания; в данном случае цикл do...while может завершиться преждевременно. Это необычное условие, поэтому убедиться в том, что цикл do...while должным образом различает корректные символы и EOF, можно с помощью static_assert.

Листинг 11.2. Использование static_assert для проверки размеров целочисленных типов

```
#include <assert.h>
#include <stdio.h>
#include <limits.h>

void clear_stdin(void) {
  int c;

  do {
    c = getchar();
    static_assert(UCHAR_MAX < UINT_MAX, "FIO34-C violation");
  } while (c != EOF);
}
```

В этом примере статическое утверждение проверяет, меньше ли максимальное значение unsigned char, UCHAR_MAX, чем максимальное значение int, UINT_MAX. Оно находится рядом с кодом, который зависит от корректности данного предположения, поэтому в случае его нарушения вы можете легко найти участок, нуждающийся в исправлении. Поскольку статические утверждения вычисляются на этапе компиляции, их наличие в исполняемом коде никак не влияет на эффективность работы программы. Более подробную информацию об этом можно найти в правиле FIO34-C стандарта CERT C (проводите различие между символами, прочитанными из файла, и EOF или WEOF).

Наконец, в листинге 11.3 static_assert используется для проверки границ на этапе компиляции. В данном фрагменте кода функция strcpy копирует константную строку prefix в статически выделенный массив str. Перед вызовом strcpy статическое утверждение позволяет убедиться в том, что в str достаточно места для сохранения по меньшей мере одного дополнительного символа с кодом ошибки.

Листинг 11.3. Использование static_assert для проверки границ

```
static const char prefix[] = "Error No: ";
#define ARRAYSIZE 14
char str[ARRAYSIZE];
```

```
// Убедимся в том, что str может вместить как минимум один
// дополнительный символ для кода ошибки
static_assert(
  sizeof(str) > sizeof(prefix),
  "str must be larger than prefix"
);
strcpy(str, prefix);
```

Это предположение может оказаться недействительным, если в ходе поддержки кода разработчик, к примеру, уменьшит `ARRAYSIZE` или поменяет префиксную строку на `"Error Number: "`. Благодаря наличию статического утверждения разработчик будет предупрежден о данной проблеме. Помните, что строковой литерал[1] — это сообщение для тех, кто пишет или сопровождает код, а не для конечных пользователей системы. Литерал должен предоставлять информацию, которая может пригодиться для отладки.

Утверждения времени выполнения

Макрос `assert` внедряет в программы диагностические проверки времени выполнения. Он определен в заголовочном файле `<assert.h>` и принимает в качестве единственного аргумента скалярное выражение:

```
#define assert(скалярное_выражение) /* зависит от реализации */
```

Макрос `assert` определяется реализацией. Если скалярное выражение равно 0, то он обычно разворачивается в инструкцию вывода информации о неудачном вызове (с текстом аргумента, именем исходного файла `__FILE__`, номером исходной строчки `__LINE__` и именем функции `__func__`, в которой этот макрос находится) в стандартный поток ошибок `stderr`. После записи данной информации в `stderr` макрос `assert` вызывает функцию `abort`.

Функция `dup_string`, показанная в листинге 11.4, использует утверждения времени выполнения, чтобы убедиться в том, что аргумент `size` меньше или равен `LIMIT` и `str` не является нулевым указателем.

Листинг 11.4. Использование assert для проверки условий программы

```
void *dup_string(size_t size, char *str ) {
  assert(size <= LIMIT);
  assert(str != NULL);
  //---snip---
}
```

[1] Второй аргумент static_assert. — *Примеч. науч. ред.*

Сообщения, выводимые этими утверждениями, могут иметь следующий вид:

```
Assertion failed: size <= LIMIT, function dup_string, file foo.c, line 122.
Assertion failed: str != NULL, function dup_string, file foo.c, line 123.
```

Предполагается, что перед использованием функции `dup_string` вызывающая сторона проверяет корректность аргументов. Убедиться в действительности этого предположения во время разработки и тестирования можно с помощью утверждений времени выполнения.

В сообщении нарушенного утверждения нередко выводится его предикатное выражение. Это позволяет применить `&&` к строковому литералу и предикату утверждения, чтобы сгенерировать дополнительную отладочную информацию. Данный подход совершенно безопасен, поскольку строковые литералы в языке C не могут иметь значение нулевого указателя. Например, мы можем переписать утверждения из листинга 11.4 так, чтобы в случае нарушения они предоставляли дополнительный контекст, как показано ниже (листинг 11.5).

Листинг 11.5. Использование assert с дополнительной контекстной информацией

```
void *dup_string(size_t size, char *str ) {
  assert(size <= LIMIT && "size is larger than the expected limit");
  assert(str != NULL && "the caller must ensure str is not null");
  //---snip---
}
```

Перед вводом кода в эксплуатацию утверждения следует отключить путем определения макроса `NDEBUG` (обычно для этого компилятору передается специальный флаг). Если в момент подключения `<assert.h>` в исходном файле уже объявлен макрос `NDEBUG`, то макрос `assert` определяется следующим образом:

```
#define assert(ignore) ((void)0)
```

Если бы макрос разворачивался в пустую строку, такой код, как:

```
assert(thing1) // пропущена точка с запятой
assert(thing2);
```

компилировался бы только в режиме выпуска (release), но не в режиме отладки. Макрос `assert` разворачивается в `((void) 0)`, а не в `0`, чтобы предотвратить предупреждения о бесполезных операторах. Макрос пере-

определяется в соответствии с текущим состоянием NDEBUG при каждом подключении <assert.h>.

Используйте статические утверждения для проверки предположений, которые можно оценить во время компиляции. Утверждения времени выполнения подходят для обнаружения некорректных предположений во время тестирования. Поскольку такие утверждения обычно отключаются перед вводом кода в эксплуатацию, их не следует задействовать для проверки условий, которые могут возникать в ходе нормальной работы, включая следующие:

- некорректный ввод;
- ошибки открытия, чтения или записи потоков ввода/вывода;
- нехватку памяти при вызове функций для динамического выделения;
- ошибки системных вызовов;
- неправильные права доступа.

В подобных условиях вместо утверждений следует использовать обычный код для проверки ошибок, который всегда присутствует в исполняемом файле. С помощью утверждений нужно проверять только предусловия, постусловия и инварианты, внедренные в код (ошибки программирования).

Параметры и флаги компиляторов

Компиляторы зачастую не включают средства оптимизации и улучшения безопасности по умолчанию. Для этого можно применять флаги сборки (Веймер, 2018). В следующем разделе я порекомендую конкретные флаги для GCC, Clang и Visual C++, но сначала поговорим о том, как ими пользоваться и зачем они нужны.

Флаги сборки нужно выбирать в соответствии с тем, чего вы пытаетесь достичь. Каждый отдельный этап разработки программного обеспечения требует своего набора флагов.

- Во время *анализа* программист пытается скомпилировать свой код. На данном этапе приходится иметь дело со множеством скучных диа-

гностических сообщений, но это лучше, чем искать соответствующие проблемы во время отладки или тестирования или обнаруживать их уже после развертывания кода. На этапе анализа следует использовать параметры компилятора, которые делают диагностические сообщения максимально подробными, чтобы устранить как можно больше дефектов.

- Во время *отладки* программист обычно пытается понять, почему его код не работает, поэтому вам следует использовать флаги компилятора для вывода отладочной информации. Это сделает ваши утверждения полезными, позволит внедрить механизмы времени выполнения для выявления ошибок и сократит неизбежный цикл «редактирование — компиляция — отладка».
- Во время *тестирования* имеет смысл отключить отладочную информацию (за исключением имен символов), чтобы получить информативную трассировку стека во время сбоев, и оставить утверждения включенными. Возможно, вам также стоит начать тестирование оптимизированных сборок. Эти параметры могут использоваться и в бета-версиях продукта, чтобы изолировать любые дефекты, обнаруженные в ходе бета-тестирования.
- *Приемочное тестирование и развертывание*. Заключительный этап состоит в сборке кода для развертывания в рабочей среде. Прежде чем развертывать систему, не забудьте должным образом проверить свою конфигурацию сборки, так как использование другого набора флагов компиляции может спровоцировать новые дефекты — например, в результате выполнения более быстрого оптимизированного кода.

Теперь я перечислю конкретные флаги, которые могут пригодиться во время компиляции и разработки.

GCC и Clang

В табл. 11.1 перечислены рекомендованные флаги компиляции и компоновки для GCC и Clang с описанием различий между этими двумя компиляторами. Документацию для этих флагов можно найти в справочнике по GCC (https://gcc.gnu.org/onlinedocs/gcc/Invoking-GCC.html) и в руководстве

пользователя по работе с компилятором Clang (https://clang.llvm.org/docs/UsersManual.html#command-line-options).

Таблица 11.1. Рекомендованные флаги компиляции и компоновки для GCC и Clang

Флаг	Назначение
`-D_FORTIFY_SOURCE=2`	Обнаруживает переполнение буфера во время работы программы
`-fpie -Wl, -pie`	Позволяет включить полную поддержку ASLR для исполняемого файла
`-fpic -shared`	Отключает перемещение кода для разделяемых библиотек
`-g3`	Генерирует обильную отладочную информацию
`-O2`	Оптимизирует код для повышения скорости и снижения расхода памяти
`-Wall`	Включает рекомендуемые предупреждения компилятора
`-Werror`	Превращает предупреждения в ошибки
`-std=c17`	Задает стандарт языка
`-pedantic`	Выводит предупреждения, относящиеся к строгому соблюдению стандарта

Флаг оптимизации -O

Флаг `-O` в верхнем регистре отвечает за *оптимизацию компилятора*. Если не указать его в командной строке или использовать `-O0`, то большинство оптимизаций отключается. В таком случае компилятор пытается ускорить процесс сборки и упростить отладку. Уровень `-Og` оптимизирует код для отладки и пропускает многие этапы оптимизации; он лучше подходит для генерации отлаживаемого кода по сравнению с `-O0`.

Когда приложение готово к развертыванию или проходит через приемочное тестирование, его код стоит оптимизировать. Включение оптимизации заставляет компилятор улучшить производительность или размер кода ценой замедления компиляции и отсутствия возможности отлаживать программу. Флаг `-O2` выполняет почти все оптимизации, не требующие

поиска баланса между размером и скоростью, и в целом рекомендуется для кода промышленного уровня. Это также минимальный флаг, который нужен для макроса `_FORTIFY_SOURCE`. Кроме того, существует уровень оптимизации `-O3`, который может повысить скорость итогового исполняемого файла за счет увеличения его размера.

При начальной компиляции кода с помощью GCC (например, на этапе анализа) флаг `-O2` имеет смысл использовать для вывода диагностических сообщений, которые генерируются только для оптимизированных сборок. Не включайте средства обнаружения ошибок работы с памятью (см. подраздел «AddressSanitizer» на с. 293) во время анализа, поскольку механизмы, внедряемые на этапе выполнения, могут провоцировать ложные срабатывания.

На уровне `-O1` проводятся быстрые оптимизации, которые не приводят к существенному замедлению процесса сборки, что может быть полезно при тестировании.

Отладка с помощью -gуровень

Флаг `-g`*`уровень`* генерирует отладочную информацию в стандартном для операционной системы формате. Объем полученной информации зависит от *уровня* отладки. По умолчанию используется уровень `-g2`. Уровень `-g3` предусматривает дополнительные сведения, такие как определения всех макросов, присутствующих в программе. Вдобавок на данном уровне можно разворачивать макросы в отладчиках, которые поддерживают эту возможность.

Предупреждения -Wall

Флаг `-Wall` позволяет включить набор полезных предупреждений. Если вы хотите включить предупреждения, не предусмотренные в `-Wall`, то можете указать `-Wextra`.

Ошибки -Werror

Флаг `-Werror` превращает все предупреждения в ошибки и требует их исправления перед началом отладки. Он просто поощряет высокую дисциплину при написании кода.

Флаг стандартов языка -std=

С помощью флага `-std=` можно выбрать определенный стандарт языка: `c89`, `c90`, `c99`, `c11`, `c17` или `c2x`. В GCC, если не указано никаких параметров диалекта языка C, по умолчанию используется флаг `-std=gnu17`, предоставляющий определенные расширения, которые в редких случаях конфликтуют со стандартом C[1]. В Clang по умолчанию применяется `-std=gnu11`. Указывайте флаг `-std=` , если пишете переносимый код. Чтобы получить доступ к новым возможностям языка, выбирайте последний стандарт.

Предупреждения -pedantic

Флаг `-pedantic` генерирует предупреждения в случаях, когда код в какой-либо мере отклоняется от стандарта. Обычно его используют в сочетании с флагом `-std=` в целях улучшения переносимости кода.

Обнаружение переполнения буфера: -D_FORTIFY_SOURCE=2

Макрос `_FORTIFY_SOURCE` является легковесным средством обнаружения переполнения буфера в функциях, которые работают с памятью и строками. Не все виды переполнения можно выявить с помощью этого макроса, однако компиляция исходного кода с использованием флага `-D_FORTIFY_SOURCE=2` обеспечивает дополнительный уровень проверки функций, таких как `memcpy`, `memset`, `strcpy`, `strcat` и `sprintf`, которые копируют память и могут быть потенциальными источниками переполнения. Ряд этих проверок может проводиться на этапе компиляции и генерировать диагностические сообщения; другие выполняются во время работы программы и могут приводить к ошибкам времени выполнения.

Позиционная независимость: -fpie -Wl, -pie и -fpic -shared

Рандомизация размещения адресного пространства (address space layout randomization, ASLR) — это механизм безопасности, рандомизирующий пространство памяти процесса, чтобы не дать злоумышленнику найти код, который он хочет выполнить. Больше об ASLR и других мерах защиты можно узнать из книги *Secure Coding in C and C++* (Сикорд, 2013).

---

[1] Значение по умолчанию разнится от версии к версии; в последней на момент издания версии GCC 10.3.0 это все еще gnu11, а не gnu17. — *Примеч. науч. ред.*

Чтобы создавать позиционно независимые программы и иметь возможность включить ASLR для своего главного исполняемого файла, вы должны указать флаги `-fpie -Wl, -pie`. Однако некоторые перемещения, выполняемые в коде главной программы, недопустимы в разделяемых библиотеках (динамически разделяемых объектах). Чтобы избежать перемещения кода в архитектурах с поддержкой позиционно независимых разделяемых библиотек, используйте флаг `-fpic` и указывайте `-shared` при компоновке. Динамически разделяемые объекты всегда позиционно независимы и, следовательно, поддерживают ASLR.

Visual C++

Visual C++ предоставляет широкий выбор параметров компиляции, и многие из них похожи на те, которые доступны в GCC и Clang[1]. Одно заметное отличие состоит в том, что для обозначения флагов Visual C++ обычно использует символ косой черты (`/`), а не дефис (`-`). В табл. 11.2 перечислены рекомендуемые флаги компиляции и компоновки для Visual C++.

Таблица 11.2. Рекомендуемые флаги компиляции для Visual C++

Флаг	Назначение
`/guard:cf`	Добавляет проверки безопасности для защиты потока управления
`/analyze`	Включает статический анализ
`/sdl`	Включает средства безопасности
`/permissive-`	Определяет режим соблюдения стандартов для компилятора
`/O2`	Устанавливает второй уровень оптимизации
`/W4`	Устанавливает четвертый уровень предупреждений компилятора
`/WX`	Превращает предупреждения в ошибки

Несколько из этих параметров аналогичны тем, которые предоставляют компиляторы GCC и Clang. Флаг `/O2` обеспечивает хороший уровень оптимизации для сдаваемого в эксплуатацию кода, а `/Od` ускоряет компиляцию и упрощает отладку за счет отключения оптимизации. Флаг `/W4` обеспечивает хороший уровень предупреждений, особенно для нового

[1] Подробности о параметрах компиляции ищите на странице https://docs.microsoft.com/ru-ru/cpp/build/reference/compiler-options-listed-by-category/.

кода, и является примерным эквивалентом флага `-Wall` в GCC и Clang. Visual C++ поддерживает флаг `/Wall`, но использовать его не рекомендуется, поскольку он провоцирует большое количество ложных срабатываний. Флаг `/WX`, как и флаг `-Werror` в GCC и Clang, превращает предупреждения в ошибки. Остальные флаги рассматриваются ниже.

Проверки безопасности /guard:cf

При использовании *защиты потока управления* (control flow guard, CFG) компилятор и компоновщик вставляют в программу дополнительные проверки безопасности, которые проводятся на этапе выполнения и выявляют попытки взломать ваш код. Параметр `/guard:cf` нужно передавать как компилятору, так и компоновщику.

Статический анализ /analyze

Флаг `/analyze` включает статический анализ, который предоставляет информацию о возможных дефектах в вашем коде. Более подробно эта тема обсуждается в разделе «Статический анализ» на с. 289.

Средства безопасности: /sdl

Флаг `/sdl` включает дополнительные средства безопасности, в том числе превращает относящиеся к безопасности предупреждения в ошибки, а также содержит дополнительные методы генерации защищенного кода. Он также включает другие возможности Microsoft *SDL* (Security Development Lifecycle — жизненный цикл безопасной разработки). Флаг `/sdl` следует использовать во всех промышленных сборках, где безопасность играет важную роль.

Соответствие стандартам: /permissive-

С помощью флага `/permissive-` можно выявлять и исправлять проблемы несоответствия стандартам, улучшая тем самым корректность и переносимость кода. Этот параметр отключает разрешительные аспекты поведения и устанавливает флаг компиляции `/Zc` для строгого соответствия. В IDE данный параметр подсвечивает код, который отклоняется от стандарта.

Отладка

Я занимаюсь профессиональным программированием 37 лет. И за все это время мне лишь однажды или, может быть, дважды удалось написать программу, которая скомпилировалась и заработала правильно с первого раза. Во всех остальных случаях нужна отладка.

Отладим дефектную программу. Код, показанный в листинге 11.6, проверяет функцию `print_error`, но не возвращает ожидаемый результат.

Листинг 11.6. Вывод ошибки

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <malloc.h>

errno_t print_error(errno_t errnum) {
  rsize_t size = strerrorlen_s(errnum);
  char* msg = malloc(size);
❶ if ((msg != NULL) && (strerror_s(msg, size, errnum) != 0)) {
    fputs(msg, stderr);
    return 0;
  }
  else {
  ❷ fputs("unknown error", stderr);
    return ENOMEM;
  }
}

int main(void) {
  print_error(ENOMEM);
  exit(1);
}
```

Если запустить эту программу в Visual C++, то она выведет следующее:

```
unknown error
```

Это не тот результат, который мы ожидали от данной проверки. Макрос `ENOMEM` должен сгенерировать строку наподобие `"out of memory"`. Вывод `"unknown error"`, вероятно, означает, что был выполнен вызов `fputs` в предложении `else` ❷; это могло произойти, только если неудачно завершился вызов `malloc` или `strerror_s`. Существует также небольшая вероятность

того, что "unknown error" — это в действительности строка, возвращенная функцией strerror_s. Чтобы узнать, так ли это, мы можем заменить выходную строку ❷ чем-то другим (например, "bananarama") и выполнить проверку еще раз. После повторной компиляции и запуска программы мы увидим следующий вывод:

```
bananarama
```

Это можно считать убедительным доказательством того, что оператор else действительно выполняется, из чего следует, что результат проверки ❶ является ложным. Нам нужно убедиться в том, так ли это на самом деле.

Начинающие программисты весьма склонны отлаживать все путем добавления в код операторов print, но использовать отладчик зачастую намного продуктивнее. Чтобы начать отладку в Visual C++, выберите в меню Debug (Отладка) пункт Start Debugging (Начать отладку). Но прежде, чем это делать, необходимо создать *точку останова*, в которой отладчик должен временно остановить выполнение программы, чтобы вы могли исследовать ее состояние. Для создания точки останова в Visual C++ нужно щелкнуть слева от номера строчки, на которой вы хотите остановиться, — например, перед оператором if ❶.

Создайте точку останова в этом месте и начните отладку. Отладчик должен остановиться перед выполнением оператора if. Прежде чем продолжать, имеет смысл исследовать состояние программы, проверив значения некоторых локальных и автоматических переменных. В частности, проверьте значение msg, чтобы убедиться в успешном вызове malloc. Это значение доступно либо на вкладке Autos (Автоматические), в которой отображаются автоматические переменные, либо на вкладке Locals (Локальные) с локальными переменными. В любом случае вы должны увидеть нечто похожее на содержимое табл. 11.3.

Таблица 11.3. Значения автоматических переменных на вкладке Autos (Автоматические)

Имя	Значение	Тип
errnum	12	int
msg	0x00a56120 "Ї Ї Ї Ї Ї Ї Ї Ї Ї Ї Ї Ї Ї Ї Ї Ї ї ї ї ї ї \bP¥"	char *
size	16	unsigned int

На этой вкладке можно видеть значения и типы трех автоматических переменных errnum, msg и size. Обратите внимание на то, что msg имеет

действительный адрес, указывающий на некий неинициализированный участок памяти. На данном этапе значения всех этих переменных выглядят адекватно.

Дальше мы можем сделать *один шаг вперед* — то есть выполнить целиком текущую строчку кода. В Visual C++ доступно три варианта пошагового выполнения: Step Into (Шаг с заходом), Step Over (Шаг с обходом) и Step Out (Шаг с выходом).

Шаг с заходом возобновляет выполнение программы, пока не будет достигнута первая строчка какой-либо вызванной функции. Это позволяет заходить внутрь вызовов при условии, что нам доступен их исходный код. *Шаг с обходом* продолжает выполнение до следующей строчки в текущей функции. *Шаг с выходом* продолжает выполнение, пока текущая функция не завершится.

В данном случае мы выберем Step Over (Шаг с обходом), чтобы увидеть, какой оператор выполняется следующим. Поток выполнения переходит к вызову `fputs` в операторе `else` ❶, но все еще непонятно почему. Можно предположить, что функция `strerror_s` по какой-то причине завершается неудачей. Попробуем выявить эту ошибку.

В Visual C++ есть возможность проверить значение, возвращаемое функцией, но вместо этого мы временно добавим в `print_error` сохранение возвращаемого из `strerror_s` значения в автоматическую переменную `status`, как показано в листинге 11.7. Теперь мы можем легко просмотреть содержимое данной переменной.

Листинг 11.7. Модифицированная функция print_error

```
errno_t print_error(errno_t errnum) {
  rsize_t size = strerrorlen_s(errnum);
  char* msg = malloc(size);
  if (msg != NULL) {
    errno_t status = strerror_s(msg, size, errnum);
  ❶ if (status != 0) {
     fputs(msg, stderr);
     return ENOMEM;
    }
  }
  else {
    fputs("unknown error", stderr);
    return ENOMEM;
  }
}
```

Создайте точку останова в строке ❶ с условием `status != 0` и проверьте в отладчике значение автоматической переменной `status`, чтобы определить, какая ошибка произошла. Вы должны увидеть, что переменная `status` равна `0`; это говорит о том, что никакой ошибки не произошло, поэтому проблема точно не вызвана сбоем в `strerror_s`.

Переменная `status` со значением `0` явно указывает на то, что проверка в строчке ❶ является инвертированной и сообщение об ошибке нужно выводить при успешном, а не неудачном выполнении `strerror_s`. Чтобы исправить этот дефект, нужно проверить, равна ли переменная `status` нулю, как показано ниже:

```
if (status == 0) {
```

Теперь, когда дефект, скорее всего, устранен, а наша уверенность в себе восстановлена, запустим эту программу еще раз:

```
Not enough spac
```

Похоже, программа следует по нужному нам пути и получает правильное сообщение об ошибке, но текст этого сообщения выглядит обрезанным. Хоть какой-то прогресс. В подобные моменты я обычно отчаиваюсь настолько, что начинаю читать документацию. Раздел K.3.7.4.2 стандарта C («Функция strerror_s») гласит следующее:

> «Если длина требуемой строки меньше чем maxsize, то строка копируется в массив, на который указывает s. В противном случае, если maxsize больше нуля, из строки в массив, на который указывает s, копируется maxsize – 1 символов, после чего элементу s[maxsize – 1] присваивается нуль-символ. Затем, если maxsize больше 3, элементам s[maxsize – 2], s[maxsize – 3] и s[maxsize – 4] присваивается точка (.)».

Теперь мы видим, что все намного хуже, чем казалось вначале (обычное явление). Поскольку интерфейсы с проверкой ограничений обычно выдают ошибку, если им не удается полностью скопировать строку, функция `print_error` предполагает, что вызов `strerror_s` завершается неудачно в связи с тем, что строка не помещается в доступный блок памяти. Но это, по всей видимости, не так. Вдобавок мы не наблюдаем задокументированное поведение — в конце строки нет `"..."`, как того требует стандарт. Согласно приведенной выше цитате мы должны были бы получить следующий вывод:

```
Not enough s...
```

Если взглянуть на спецификацию функции `strerrorlen_s`, то можно заметить, что она возвращает количество символов в строке с полным сообщением (без учета нуль-символа). Данное объяснение звучит логично и соответствует поведению функции `strlen`. Поэтому проблему, наверное, можно решить путем прибавления к `size` одного байта:

```
rsize_t size = strerrorlen_s(errnum) + 1;
```

Еще раз дадим волю своей самоуверенности и запустим программу:

```
Not enough space
```

Похоже, наша уверенность в своих силах была оправданной, так как программа успешно вывела полное сообщение об ошибке. Тем не менее в этой версии `print_error` по-прежнему остается один дефект. Позже мы к нему еще вернемся, но попробуйте найти его самостоятельно.

Успешно отладив функцию `print_error`, вы можете вернуть ее к корректному и более компактному виду или оставить как есть. Если вы снова отредактируете код, то не забудьте еще раз проверить, по-прежнему ли он работает. Не жалейте времени на то, чтобы сообщать разработчикам о дефектах в их реализации.

Модульное тестирование

Итак, у нас есть «рабочая» реализация функции `print_error`. Пришло время провести модульное тестирование, чтобы убедиться в ее работоспособности. *Модульные тесты* — небольшие программы, выполняющие ваш код. *Модульное тестирование* — процесс проверки того, что каждый модуль вашего программного обеспечения работает так, как было задумано. *Модуль* — наименьший участок кода, который можно протестировать; в языке C эту роль обычно играет отдельная функция или абстракция данных.

Вы можете писать тесты, которые напоминают код обычного приложения (как, например, в листинге 11.7), но обычно для этого лучше использовать *фреймворк модульного тестирования*. Таких фреймворков существует несколько, включая Google Test, CUnit, Unity, DejaGnu и CppUnit. Мы рассмотрим самый популярный из них (согласно последнему исследованию экосистемы разработки на C, проведенному компанией JetBrains): Google Test.

Google Test работает в Linux, Windows и macOS. Тесты нужно писать на C++, поэтому у вас будет возможность выучить еще один (хоть и родственный) язык программирования. Чтобы проверить поведение тестируемого кода, создаются утверждения. В Google Test они представляют собой функциональные макросы и являются настоящим языком написания тестов. В случае сбоя или невыполнения утверждения тест проваливается, во всех остальных случаях считается успешным. Результатом утверждения может быть либо успешное выполнение, либо критический или некритический отказ. При возникновении критического отказа работа текущей функции прерывается; в противном случае программа продолжает работать.

Фреймворк Google Test интегрирован в среду Visual Studio 2017 и более новые версии в качестве стандартного компонента разработки для настольных компьютеров на языке C++. Это упрощает его настройку и использование в Windows[1].

Мы воспользуемся Google Test в Visual Studio, чтобы протестировать функцию `get_error`, представленную в листинге 11.8. Она похожа на `print_error`, но, вместо того чтобы выводить строку с сообщением, которая соответствует номеру ошибки, переданному ей в качестве аргумента, она ее просто возвращает.

Листинг 11.8. Функция get_error

```
char *get_error(errno_t errnum) {
  rsize_t size = strerrorlen_s(errnum) + 1;
  char* msg = malloc(size);
  if (msg != NULL) {
    errno_t status = strerror_s(msg, size, errnum);
    if (status != 0) {
      strncpy_s(msg, size, "unknown error", size-1);
    }
  }
  return msg;
}
```

В листинге 11.9 показан тест для этой функции. Код на C++ в основном является заготовкой, и его можно копировать без изменений. Это в том числе относится и к функции `main`, вызывающей функциональный макрос `RUN_ALL_TESTS`, который выполняет определенные вами тесты.

[1] О том, как добавить и сконфигурировать проект Google Test в Visual C++, можно прочесть на странице https://docs.microsoft.com/ru-ru/visualstudio/test/how-to-use-google-test-for-cpp/.

Листинг 11.9. Модульные тесты для функции get_error

```
#include "pch.h"
❶ extern "C" char* get_error(errno_t errnum); // реализована в исходном файле C

namespace {
❷ TEST(MyTestSuite, MsgTestCase) {
    EXPECT_STREQ(get_error(ENOMEM), "Not enough space");
    EXPECT_STREQ(get_error(ENOTSOCK), "Not a socket");
    EXPECT_STREQ(get_error(EPIPE), "Broken pipe");
  }
} // пространство имен
int main(int argc, char** argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Здесь есть два участка, которые не относятся к заготовке: объявление `extern "C"` ❶ и сам тест ❷. Объявление `extern "C"` меняет требования к компоновке, чтобы компоновщик C++ не декорировал имя функции, как он обычно делает. Вам нужно добавить аналогичное объявление для каждой тестируемой функции, или же вы можете просто подключить заголовочный файл C внутри блока `extern "C"`, как показано ниже:

```
extern "C" {
  #include "api_to_test.h"
}
```

В данном тесте используется макрос `TEST`, который определяет конкретный тестовый кейс и принимает два аргумента. Первым аргументом выступает название *набора тестов* (совокупности всех тестовых случаев, выполняемых в отдельно взятом цикле тестирования). Второй аргумент — это имя *тестового случая*, который представляет собой набор предусловий, входных значений, действий (когда это применимо), ожидаемых результатов и постусловий, разработанных с учетом условий теста. Определения других терминов можно найти на сайте Международной квалификационной комиссии по тестированию программного обеспечения (https://glossary.istqb.org/).

Все утверждения Google Test вместе с любыми дополнительными операторами C++, которые вы хотите включить, должны находиться в теле функции. В листинге 11.9 (см. выше) используется макрос `EXPECT_STREQ`, который проверяет, имеют ли две строки одинаковое содержимое. Мы применили утверждения для нескольких номеров ошибок, чтобы

убедиться в том, что для каждого из них функция возвращает правильную строку. Утверждение `EXPECT_STREQ` является некритическим, поскольку тестирование может продолжаться даже в случае его невыполнения. Подобные утверждения считаются более предпочтительными по сравнению с критическими, так как с их помощью можно выявить и исправить сразу несколько дефектов за один цикл «запуск — редактирование — компиляция». Если тестирование нельзя продолжать после первого же отказа (например, когда последующие операции зависят от предыдущего результата), то вы можете использовать критическое утверждение `ASSERT_STREQ`.

Этот тестовый случай проверяет несколько номеров ошибок из `<errno.h>`. То, сколько из них необходимо протестировать, зависит от того, чего вы пытаетесь достичь. В идеале тесты должны быть всеобъемлющими; то есть утверждение нужно предусмотреть для каждого номера ошибки в `<errno.h>`. Но это может быть утомительно; убедившись в том, что ваш код работает, вы в основном начинаете проверять корректность реализации функций из стандартной библиотеки C, которые вы используете в тестируемой функции. Вместо этого можно протестировать номера ошибок, которые вы с большей вероятностью будете встречать, но это тоже может потребовать много усилий, поскольку вам придется пройтись по всем функциям в своей программе и определить, какие коды ошибок они могут вернуть.

В листинге 11.9 мы реализовали выборочную проверку: несколько случайно выбранных номеров из разных частей списка. Результат выполнения этого теста показан в листинге 11.10.

Листинг 11.10. Тестовый прогон MyTestSuite.MsgTestCase

```
.\crash-test.exe
[==========] Running 1 test from 1 test case.
[----------] Global test environment setup.
[----------] 1 test from MyTestSuite
[ RUN ] MyTestSuite.MsgTestCase
crash-test\TestMain.cpp(39): error: Expected equality of these values:
  get_error(128)
❶  Which is: "Unknown error"
❷  "Not a socket"
[ FAILED ] MyTestSuite.MsgTestCase (5 ms)
[----------] 1 test from MyTestSuite (10 ms total)

[----------] Global test environment tear-down
```

```
[==========] 1 test from 1 test case ran. (31 ms total)
[ PASSED ] 0 tests.
[ FAILED ] 1 test, listed below:
[ FAILED ] MyTestSuite.MsgTestCase

1 FAILED TEST
```

Два утверждения выполняются, но вызов `get_error(ENOTSOCK)` проваливается, поскольку `strerror_s` возвращает `"Unknown error"` ❶ вместо `"Not a socket"` ❷. Это несколько неожиданно (по крайней мере я такого не ожидал).

Оформив полагающийся отчет о программном дефекте, мы все еще не знаем, как решить данную проблему. Мы можем попросту с ней смириться. Это будет означать, что в случае возвращения `errno` пользователь увидит загадочное сообщение `"Unknown error"` вместо чего-то более полезного. Если вас это устраивает, то можете заменить ожидаемый результат строкой `"Unknown error"` или оставить тест как есть, помня о том, что рано или поздно вам придется отправить в эксплуатацию приложения с проваленным тестовым случаем. Как бы то ни было, у вас будет возможность узнать, исправил ли разработчик дефект, отчет о котором вы ему отправили.

Если вам не по душе малоинформативные сообщения об ошибках, то можете написать обертку вокруг функции `strerror_s`, которая предоставляет недостающие строки. В этом случае вам, наверное, стоит провести всеобъемлющее тестирование функций `get_error` и `strerror_s`, чтобы определить все варианты, нуждающиеся в исправлении.

Статический анализ

К *статическому анализу* относят любой процесс исследования кода без его выполнения (ISO/IEC TS 17961:2013) в целях предоставления информации о потенциальных программных дефектах. Такой анализ можно выполнить вручную, хотя по мере усложнения программы данный подход быстро становится непрактичным. Вместо этого можно использовать специальные инструменты.

Статический анализ имеет практические ограничения, поскольку корректность программного обеспечения не поддается вычислению. Например, проблема остановки из теории алгоритмов гласит, что существуют такие

программы, управляющую логику которых невозможно точно определить статически. В результате для ряда программ нельзя вычислить свойства, которые зависят от управляющей логики. Из этого следует, что статический анализ может пропускать реальные проблемы и в то же время выявлять те, которых не существует.

Неспособность выявить реальный дефект в коде называют *ложноотрицательным результатом*. Это серьезный недостаток анализа, который может вызвать у вас ложное чувство защищенности. Большинство инструментов проявляют излишнюю осторожность, что приводит к ложным срабатываниям. *Ложное срабатывание* — это *ложноположительный* результат тестирования, который ошибочно сообщает о наличии проблемы. В попытках избежать чрезмерного количества ложных срабатываний инструмент может сообщить о дефектах, связанных с повышенным риском, и проигнорировать другие. Ложные срабатывания также могут встречаться, когда код слишком сложен для полноценного анализа. Вероятность этого повышается при использовании библиотек и указателей на функции.

В идеале результаты, выдаваемые инструментом, должны быть полными и надежными. Анализатор считается *надежным (sound)*, если никогда не выдает ложноотрицательных результатов, и *полным*, если у него не бывает ложных срабатываний. Возможные сочетания показаны на рис. 11.1.

		Ложноположительные: Да	Ложноположительные: Нет
Ложноотрицательные	Да	Неполные и ненадежные	Полные и ненадежные
	Нет	Неполные и надежные	Полные и надежные

Рис. 11.1. Полнота и надежность

В процессе компиляции проводится ограниченный статический анализ, диагностирующий строго локализованные проблемы, которые не требуют особых объяснений. Например, при сравнении знакового значения с беззнаковым компилятор может выдать предупреждение о несоответствии типов, поскольку для обнаружения этой ошибки не нуждается ни в какой дополнительной информации. Управлять выводом, который отображает

компилятор, можно с помощью уже знакомых вам флагов: `/W4` для Visual C++ и `-Wall` для GCC и Clang.

В целом компиляторы предоставляют высококачественные результаты диагностики, и их не стоит игнорировать. Вместо того чтобы заглушать предупреждения за счет приведения типов или вносить произвольные изменения до тех пор, пока они не исчезнут, всегда пытайтесь понять причину их возникновения и пытайтесь ее устранить. Более подробную информацию об этом можно найти в правиле MSC00-C стандарта CERT C (код должен компилироваться без замечаний даже с высокими уровнями предупреждений).

Разобравшись с предупреждениями, которые компилятор выдает для вашего кода, вы можете воспользоваться отдельным статическим анализатором для поиска дополнительных изъянов. Статические анализаторы умеют диагностировать более сложные дефекты, вычисляя выражения в вашей программе, глубоко исследуя управляющую логику и потоки данных, оценивая возможные диапазоны значений и проверяя потенциальные пути выполнения кода.

Поиск и определение ошибок в программе с помощью специального инструмента может сэкономить вам долгие часы, которые были бы потрачены на тестирование и отладку, и избежать убытков, связанных с развертыванием дефектного кода. Существует широкий выбор бесплатных и коммерческих средств статического анализа. Например, в Visual C++ есть статический анализатор, который можно включить с помощью флага `/analyze`; он позволяет выбирать отдельные или сразу все наборы правил, которые нужно проверить (они разделены на такие категории, как «рекомендованные», «безопасность» и «интернационализация»). Больше об этом можно узнать на сайте Microsoft (https://docs.microsoft.com/ru-ru/visualstudio/code-quality/code-analysis-for-c-cpp-overview/). У Clang тоже есть статический анализатор, который можно использовать в виде отдельного инструмента или внутри Xcode (https://clang-analyzer.llvm.org/). В GCC 10 тоже появятся элементарные средства статического анализа. Кроме того, существуют коммерческие продукты, такие как CodeSonar от GrammaTech, TrustInSoft Analyzer, SonarQube от SonarSource, Coverity от Synopsys, LDRA Testbed, Helix QAC от Perforce и др.

Многие средства статического анализа имеют возможности, которых нет у других, поэтому будет разумно использовать сразу несколько из них.

Динамический анализ

Динамический анализ — это процесс оценивания системы или компонента во время выполнения. Для этого используются и другие аналогичные термины, такие как *анализ времени выполнения*.

Один из распространенных подходов к динамическому анализу состоит в *инструментировании* кода — например, путем включения флагов компиляции, которые внедряют в исполняемый файл дополнительные инструкции, и выполнения этой видоизмененной программы. Аналогичная методика применяется в библиотеке для отладки выделения памяти `dmalloc`, описанной в главе 6. Библиотека `dmalloc` заменяет вызовы, предназначенные для управления памятью, собственными, у которых есть средства отладки с возможностью изменения конфигурации. Вы можете отслеживать эти вызовы с помощью утилиты командной строки (которая тоже называется `dmalloc`) в поисках утечек памяти и дефектов, таких как запись за пределы объекта или использования указателя после его освобождения.

Преимущество динамического анализа состоит в том, что он редко выдает ложные срабатывания, поэтому если подобного рода инструмент нашел проблему, то ее необходимо исправить! Из недостатков можно выделить то, что динамический анализатор должен покрыть достаточный объем кода. Если дефектный код не выполняется во время тестирования, то проблема не будет выявлена. Еще один недостаток в том, что инструментирование может нежелательным образом повлиять на другие аспекты программы — например, повысить накладные расходы или увеличить размер двоичного файла.

Пример эффективного средства динамического анализа — AddressSanitizer. Этот инструмент доступен (бесплатно) для разных компиляторов. У него есть несколько родственных инструментов, включая ThreadSanitizer, MemorySanitizer, Hardware-Assisted AddressSanitizer, and UndefinedBehaviorSanitizer[1]. Существует множество других средств динамического анализа, как коммерческих, так и бесплатных. Чтобы продемонстрировать их полезность, я подробно расскажу о AddressSanitizer.

[1] Подробности об этих анализаторах ищите на странице https://github.com/google/sanitizers/.

AddressSanitizer

AddressSanitizer (ASan) — средство обнаружения ошибок динамической памяти для программ, написанных на C и C++; см. https://github.com/google/sanitizers/wiki/AddressSanitizer. Оно интегрировано в LLVM 3.1 и GCC 4.8, а также в более новые версии этих компиляторов. Поддержка ASan есть и в Visual Studio, начиная с версии 2019 года. Этот динамический анализатор умеет находить целый ряд шибок памяти, включая следующие:

- использование после освобождения (разыменование висячего указателя);
- переполнение кучи, стека и глобального буфера;
- применение после возвращения;
- использование после выхода из области видимости;
- неправильный порядок инициализации;
- утечки памяти.

Чтобы увидеть, какую пользу может принести этот инструмент, мы задействуем его для инструментирования видоизмененной функции `print_error` из листинга 11.7 и функции `get_error` из листинга 11.8, а затем проанализируем полученный код в Ubuntu Linux. С этой целью мы расширим уже написанные нами тесты для `get_error` (см. листинг 11.9).

Код Google Test в листинге 11.11 проверяет две служебные функции для вывода ошибок. Помимо проверки того, возвращает ли функция `get_error` корректную строку, мы добавили некритическое утверждение `EXPECT_EQ`, которое проверяет, возвращает ли вызов `print_error` значение `0`.

Листинг 11.11. Проверка вывода ошибок

```
TEST(PrintTests, MsgTestCase) {
  ASSERT_STREQ(get_error(ENOMEM), "Not enough space");
  ASSERT_STREQ(get_error(ENOTSOCK), "Not a socket");
  ASSERT_STREQ(get_error(EPIPE), "Broken pipe");
  EXPECT_EQ(print_error(ENOMEM), 0);
  EXPECT_EQ(print_error(ENOTSOCK), 0);
  EXPECT_EQ(print_error(EPIPE), 0);
}
```

Теперь соберем и выполним этот код в Ubuntu Linux.

Сборка кода в Ubuntu Linux

Если вам удалось скомпилировать и протестировать функцию `get_error` из листинга 11.8, то на вашем компьютере с Ubuntu Linux уже должна быть готовая реализация интерфейса с проверкой ограничений. Теперь вам нужно установить в Ubuntu пакет разработки Google Test с помощью команды `apt-get`:

```
% sudo apt-get install libgtest-dev
```

Данный пакет содержит только исходный код, поэтому вам нужно его скомпилировать, чтобы получить необходимые библиотечные файлы. Перейдите в каталог /usr/src/gtest, в котором должны находиться исходники, и используйте следующие команды, чтобы скомпилировать библиотеку с помощью `cmake`:

```
% sudo apt-get install cmake # install cmake
% cd /usr/src/gtest
% sudo cmake CMakeLists.txt
% sudo make
% # copy or symlink libgtest.a and libgtest_main.a to your /usr/lib folder
% sudo cp *.a /usr/lib
```

Теперь, прежде чем продолжать, нужно собрать и выполнить программу `PrintTests` из листинга 11.11. Для этого, скорее всего, придется внести изменения в исходный код и файлы сборки.

Выполнение тестов

Вызов `strerror_s`, который применяется в функциях `get_error` и `print_error`, возвращает строку с сообщением в соответствии с региональными настройками. При выполнении тестов из листинга 11.11 вы, наверное, заметите что-то неладное, а именно, что все тесты для функции `get_error` проваливаются. Дело в том, что изначально они разрабатывались для Windows с использованием Visual C++, и в Ubuntu Linux возвращаются другие локализированные сообщения. Если это не то, чего вы ожидали, то вам, возможно, следует переписать эти две функции так, чтобы они возвращали те же строки вне зависимости от региональных настроек. Мы также можем отредактировать тесты (листинг 11.12), чтобы они проверяли возвращение локализированных строк с сообщениями (выделенных жирным шрифтом).

Листинг 11.12. Видоизмененные проверки вывода ошибок

```
TEST(PrintTests, MsgTestCase) {
  EXPECT_STREQ(get_error(ENOMEM), "Cannot allocate memory");
  EXPECT_STREQ(get_error(ENOTSOCK), "Socket operation on non-socket");
  EXPECT_STREQ(get_error(EPIPE), "Broken pipe");
  EXPECT_EQ(print_error(ENOMEM), 0);
  EXPECT_EQ(print_error(ENOTSOCK), 0);
  EXPECT_EQ(print_error(EPIPE), 0);
}
```

Выполнение переписанных тестов из листинга 11.12 должно иметь положительные результаты (листинг 11.13). При взгляде на них неопытный тестировщик может подумать: «А код-то работает!» Но, чтобы убедиться в отсутствии в вашем коде дефектов, необходимо предпринять дополнительные шаги.

Листинг 11.13. Тестовый прогон PrintTests

```
student@scode:~/Examples/asan$ ./runTests
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from PrintTests
[ RUN ] PrintTests.MsgTestCase
Cannot allocate memory
Socket operation on non-socket
Broken pipe
[       OK ]
PrintTests.MsgTestCase (0 ms)
[----------] 1 test from PrintTests (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (0 ms total)
[ PASSED ] 1 test.
```

Итак, мы создали рабочую среду теста. Теперь пришло время выполнить инструментирование кода.

Инструментирование кода

Инструментировать код с помощью AddressSanitizer, при компиляции и компоновке программы можно, добавив флаг `-fsanitize=address`. Чтобы получить более информативную трассировку стека в сообщениях об ошибках, добавьте флаг `-fno-omit-frame-pointer`, а флаг `-g3` позволит вам получить информацию об отладочных символах. Если вы используете

cmake, то эти флаги можно установить путем добавления следующей строчки в файл CMakeLists.txt:

```
set (CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -g3 -fno-omit-frame-pointer
-fsanitize=address")
```

Как уже упоминалось, AddressSanitizer работает с Clang, GCC и Visual C++[1]. В зависимости от версии компилятора, которую вы используете, вам, возможно, придется определить следующие переменные среды:

```
ASAN_OPTIONS=symbolize=1
ASAN_SYMBOLIZER_PATH=/path/to/llvm_build/bin/llvm-symbolizer
```

Сделав это, попробуйте заново собрать и выполнить свои тесты.

Выполнение тестов

Модульные тесты, написанные вами с использованием Google Test, должны и дальше успешно выполняться, но при этом заставят ваш код попотеть, позволяя анализатору AddressSanitizer искать дополнительные проблемы. Вывод PrintTests должен расшириться, как показано в листинге 11.14.

Листинг 11.14. Инструментированный тестовый прогон PrintTests

```
❶ ==16447==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 31 byte(s) in 1 object(s) allocated from:
    #0 0x7fd8e3a1db50 in __interceptor_malloc
    (/usr/lib/x86_64-linux-gnu/libasan.so.4+0xdeb50)
  ❷ #1 0x564622aa0b39 in print_error ~/asan/PrintUtils/print_utils.c:12
    #2 0x564622a65839 in TestBody ~/asan/TestMain.cpp:49
    #3 0x564622a91754 in void
       testing::internal::HandleSehExceptionsInMethodIfSupported
       <testing::Test, void>(testing::Test*, void (testing::Test::*)(),
       char const*) (~/asan/runTests+0x35754)
    #4 0x564622a8b75c in void
       testing::internal::HandleExceptionsInMethodIfSupported
       <testing::Test, void>(testing::Test*, void (testing::Test::*)(),
       char const*) (~/asan/runTests+0x2f75c)
    #5 0x564622a6f139 in testing::Test::Run() (~/asan/runTests+0x13139)
    #6 0x564622a6fa6f in testing::TestInfo::Run() (~/asan/runTests+0x13a6f)
```

[1] См. AdressSanitizer (ASan) for Windows with MVSC в блоге разработчиков Microsoft, посвященном C++ (https://devblogs.microsoft.com/cppblog/addresssanitizer-asan-for-windows-with-msvc/).

```
    #7 0x564622a700f9 in testing::TestCase::Run() (~/asan/runTests+0x140f9)
    #8 0x564622a76fc5 in testing::internal::UnitTestImpl::RunAllTests()
       (~/asan/runTests+0x1afc5)
    #9 0x564622a9291a in bool
       testing::internal::HandleSehExceptionsInMethodIfSupported
       <testing::internal::UnitTestImpl, bool>(testing::internal::UnitTest
       Impl*, bool (testing::internal::UnitTestImpl::*)(), char const*)
       (~/asan/runTests+0x3691a)
    #10 0x564622a8c598 in bool
        testing::internal::HandleExceptionsInMethodIfSupported
        <testing::internal::UnitTestImpl, bool>(testing::internal::UnitTest
        Impl*, bool (testing::internal::UnitTestImpl::*)(), char const*)
        (~/asan/runTests+0x30598)
    #11 0x564622a75b89 in testing::UnitTest::Run() (~/asan/runTests+0x19b89)
    #12 0x564622a66715 in RUN_ALL_TESTS() /usr/include/gtest/gtest.h:2233
    #13 0x564622a65fd7 in main ~/asan/TestMain.cpp:57
    #14 0x7fd8e2b13b96 in __libc_start_main
        (/lib/x86_64-linux-gnu/libc.so.6+0x21b96)
```

В листинге 11.14 показана лишь первая найденная проблема. Большая часть данной трассировки стека относится к самой инфраструктуре теста и не помогает в поиске дефектов, поэтому нас не интересует. Вся полезная информация находится в верхней части листинга (и стека).

Прежде всего мы видим, что `LeakSanitizer`, компонент AddressSanitizer, «обнаружил утечки памяти» ❶ и что это прямая утечка 31 байта из одного объекта. Трассировка стека указывает на следующую строчку кода ❷:

```
#1 0x564622aa0b39 in print_error ~/asan/PrintUtils/print_utils.c:12
```

В этой строчке, принадлежащей функции `print_error`, происходит вызов `malloc`:

```
errno_t print_error(errno_t errnum) {
  rsize_t size = strerrorlen_s(errnum);
  char* msg = malloc(size);
  //---snip---
}
```

Это довольно очевидная ошибка; значение, возвращаемое из `malloc`, присваивается автоматической переменной, которая определена в области видимости функции `print_error` и никогда не освобождается. Мы теряем возможность освободить динамически выделенную память после того, как функция завершается и время жизни объекта, хранящего указатель на выделенную память, заканчивается. Чтобы исправить данную проблему, сделайте вызов `free(msg)` после того, как выделенный участок больше

не нужен, но перед завершением функции. Продолжайте выполнять тесты и исправлять любые обнаруженные дефекты, пока вас не удовлетворит качество вашей программы. Вы, конечно, можете развернуть свой код в пятницу и отключить телефон на выходных, но стоит ли?

Упражнения

Попробуйте самостоятельно выполнить следующие упражнения.

1. Исследуйте дефектный код из листинга 11.1 с помощью статического анализатора, встроенного в Visual C++. Удалось ли вам найти что-то новое?
2. Исследуйте остальные результаты теста `PrintTests`, инструментированного с использованием AddressSanitizer. Устраните оставшиеся реальные ошибки.
3. Попробуйте инструментировать тест `PrintTests` с помощью анализаторов, доступных по адресу https://github.com/google/sanitizers/, и попробуйте исправить любые найденные проблемы.
4. Примените эти и другие похожие методы тестирования, отладки и анализа к своим реальным проектам.

Резюме

В данной главе вы познакомились со статическими утверждениями и утверждениями времени выполнения, а также с одними из важнейших и наиболее рекомендуемых флагов компиляции для GCC, Clang и Visual C++. Вы научились отлаживать, тестировать и анализировать свой код с помощью статических и динамических анализаторов. Это важные заключительные уроки, поскольку, работая профессиональным программистом на C, существенную часть своего времени вы будете тратить на отладку и анализ кода.

Список литературы

- American National Standards Institute (ANSI). Information Systems — Coded Character Sets — 7-Bit American National Standard Code for Information Interchange (7-Bit ASCII). 1986. ANSI X3.4-1986.
- *Boute R. T.* The Euclidean Definition of the Functions div and mod // ACM Transactions on Programming Language and Systems 14, 1992. № 2 (April): P. 127–144. http://dx.doi.org/10.1145/128861.128862.
- *Dijkstra E.* Go To Statement Considered Harmful // Communications of the ACM 11, 1968. № 3. https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf.
- *Hollasch S.* IEEE Standard 754 Floating-Point Numbers. 2018/ https://steve.hollasch.net/cgindex/coding/ieeefloat.html.
- *Hopcroft J. E., Ullman J. D.* Introduction to Automata Theory, Languages and Computation // Reading, MA: Addison-Wesley, 1979.
- IEEE and The Open Group. 2008. Standard for Information Technology — Portable Operating System Interface (POSIX), Base Specifications. Issue 7. IEEE Std 1003.1, 2018 edition.
- IEEE. 2008. IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2008 (August): 1–70. https://ieeexplore.ieee.org/document/4610935.
- ISO/IEC/IEEE. 2011. Information Technology — Microprocessor Systems — Floating-Point Arithmetic. ISO/IEC/IEEE 60559:2011. https://www.iso.org/obp/ui/#iso:std:57469:en.
- ISO/IEC. 1990. Programming Languages — C. 1st ed. ISO/IEC 9899:1990.
- ISO/IEC. 1999. Programming Languages — C. 2nd ed. ISO/IEC 9899:1999.
- ISO/IEC. 2007. Information Technology — Programming Languages, Their Environments and System Software Interfaces — Extensions to the C Library — Part 1: Bounds-Checking Interfaces. ISO/IEC TR 24731-1:2007.
- ISO/IEC. 2010. Information Technology — Programming Languages, Their Environments and System Software Interfaces — Extensions to the

C Library — Part 2: Dynamic Allocation Functions. ISO/IEC TR 24731-2:2010.

- ISO/IEC. 2011. Programming Languages — C. 3rd ed. ISO/IEC 9899:2011.
- ISO/IEC. 2013. Information Technology — Programming Languages, Their Environments and System Software Interfaces — C Secure Coding Rules. ISO/IEC TS 17961:2013.
- ISO/IEC. 2014. Floating-Point Extensions for C — Part 1: Binary Floating-Point Arithmetic. ISO/IEC TS 18661-1:2014.
- ISO/IEC. 2015. Floating-Point Extensions for C — Part 3: Interchange and Extended Types. ISO/IEC TS 18661-3:2015.
- ISO/IEC. 2018. Programming Languages — C. 4th ed. ISO/IEC 9899:2018.
- *Kernighan B. W., Ritchie D. M.* The C Programming Language, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 1988. (*Керниган Б., Ритчи Д.* Язык программирования С. — М.: Вильямс, 2015. — 304 с.)
- *Knuth D.* Fundamental Algorithms, 3rd ed. Volume 1 of The Art of Computer Programming. Chapter 2. Pages 438–442. Boston: Addison-Wesley, 1997. (*Кнут Д. Э.* Искусство программирования. Том 1. Основные алгоритмы. 3-е изд. — М.: Вильямс, 2002. Т. 1. — 720 с.)
- *Kuhn M.* UTF-8 and Unicode FAQ for Unix/Linux. June 4, 1999. https://www.cl.cam.ac.uk/~mgk25/unicode.html.
- *Lamport L.* How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs // IEEE Transactions on Computers C-28, 1979. 9 (September). P. 690–691.
- *Lewin M.* All About XOR // Overload Journal, 2012. 109 (June). https://accu.org/index.php/journals/1915.
- *Saks D.* Tag vs. Type Names. October 1, 2002. https://www.embedded.com/electronics-blogs/programming-pointers/4024450/Tag-vs-Type-Names.
- *Schoof R.* Four Primality Testing Algorithms. 2008. arXiv preprint arXiv:0801.3840.
- *Seacord R. C.* Secure Coding in C and C++, 2nd ed. Boston: Addison-Wesley Professional, 2013.

- *Seacord R. C.* The Cert C Coding Standard: 98 Rules for Developing Safe, Reliable and Secure Systems, 2nd ed. Boston: Addison-Wesley Professional, 2014.
- *Seacord R. C.* Uninitialized Reads // Communications of the ACM 60, 2017. No. 4 (March). P. 40–44. https://doi.org/10.1145/3024920.
- *Seacord R. C.* Bounds-Checking Interfaces: Field Experience and Future Directions // NCC Group whitepaper, 2019. June. https://www.nccgroup.trust/us/our-research/bounds-checking-interfaces-field-experience-and-future-directions/.
- The Unicode Consortium. The Unicode Standard: Version 13.0 — Core Specification. Mountain View, CA: The Unicode Consortium, 2020. http://www.unicode.org/versions/Unicode13.0.0/.
- *Weimer F.* Recommended Compiler and Linker Flags for GCC // Red Hat Developer. March 21, 2018. https://developers.redhat.com/blog/2018/03/21/compiler-and-linker-flags-gcc/.

Об авторе

Роберт С. Сикорд (Robert.Seacord@nccgroup.com) занимает должность технического директора в NCC Group, где создает учебные курсы по C, C++ и другим языкам и преподает на них. Роберт также входит в экспертный состав международной рабочей группы по стандартизации языка программирования C, ISO/IEC JTC1/SC22/WG14. На его счету несколько других книг, включая *The CERT C Coding Standard, Second Edition* (Addison-Wesley, 2014), *Secure Coding in C and C++, Second Edition* (Addison-Wesley, 2013) и *Java Coding Guidelines: 75 Recommendations for Reliable and Secure Programs* (Addison-Wesley, 2014). Кроме того, Роберт опубликовал более 50 статей о безопасности программного обеспечения, компонентной разработке ПО, проектировании веб-систем, модернизации устаревших систем, репозиториях компонентов и поисковых системах, а также о проектировании и разработке пользовательского интерфейса.

О соавторе

Аарон Баллман (aaron@aaronballman.com) занимается развитием клиентской части компилятора в GrammaTech, Inc. и работает в основном над средством статического анализа CodeSonar. Он также отвечает за сопровождение клиентской части Clang — популярного компилятора с открытым исходным кодом для C, C++ и других языков. Аарон входит в экспертный состав комитетов во главе стандартов языков программирования C (JTC1/SC22/WG14) и C++ (JTC1/SC22/WG21). Его основной профессиональный интерес состоит в том, чтобы помочь программистам находить ошибки в их коде за счет совершенствования архитектуры языка, средств диагностики и инструментария. Свободное от программирования время Аарон любит проводить вместе со своей семьей.

О научном редакторе

Мартин Себор работает в Red Hat и является ведущим разработчиком в команде GNU Toolchain. Он специализируется на компиляторе GCC, в частности на обнаружении, диагностировании и предотвращении проблем безопасности в программах на языках C и C++, а также на оптимизации алгоритмов для работы со строками. До своего перехода в Red Hat в 2015 году он работал над инструментальными средствами компилятора в Cisco. Мартин является членом комитета по стандартизации C++ с 1999 года, а в 2010 году вошел в комитет по развитию языка C. Проживает вместе со своей женой в маленьком городке Лайонс, Колорадо.

Роберт С. Сикорд

Эффективный C. Профессиональное программирование

Перевел с английского *А. Павлов*

Руководитель дивизиона	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Н. Гринчик*
Научный редактор	*Р. Хазанский*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*В. Мостипан*
Корректоры	*Е. Павлович, Н. Смолич*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 08.2021. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 06.08.21. Формат 70×100/16. Бумага офсетная. Усл. п. л. 24,510. Тираж 700. Заказ 0000.