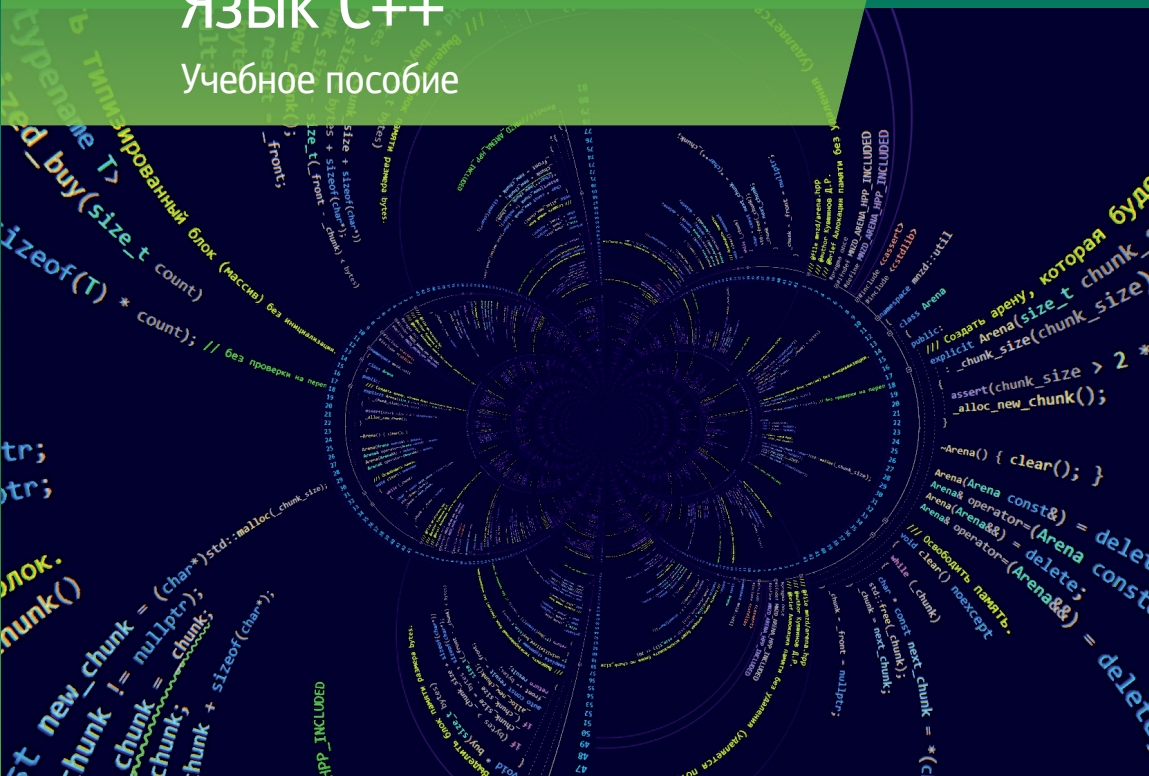
Уральский федеральный университет
имени первого Президента России Б.Н.Ельцина
Институт естественных наук и математики

Д. Р. КУВШИНОВ
С. И. ОСИПОВ

ОСНОВЫ ПРОГРАММИРОВАНИЯ
Язык C++

Учебное пособие



МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

УРАЛЬСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ
ИМЕНИ ПЕРВОГО ПРЕЗИДЕНТА РОССИИ Б. Н. ЕЛЬЦИНА

Д. Р. Кувшинов, С. И. Осипов

ОСНОВЫ ПРОГРАММИРОВАНИЯ
Язык C++

Учебное пособие

Рекомендовано методическим советом
Уральского федерального университета в качестве учебного пособия
для студентов, обучающихся по направлениям подготовки
01.03.03 «Механика и математическое моделирование»,
01.03.04 «Прикладная математика»

Екатеринбург
Издательство Уральского университета
2021

УДК 004.43(075.8)
ББК 32.973.26-018.1я73
К88

Под общей редакцией Д. Р. Кувшинова

Рецензенты:
сектор отдела динамических систем
Института математики и механики УрО РАН
(заведующий сектором, доктор физико-математических наук
А. А. Успенский);
С. С. Александрова, кандидат технических наук
(Московский авиационный институт (Национальный исследовательский университет))

Кувшинов, Д. Р.

К88 Основы программирования : язык C++ : учебное пособие / Д. Р. Кувшинов, С. И. Осипов ; под общ. ред. Д. Р. Кувшинова ; Министерство науки и высшего образования Российской Федерации, Уральский федеральный университет. — Екатеринбург : Изд-во Урал. ун-та, 2021. — 490 с. : ил. — Библиогр.: с. 488–489. — 30 экз. — ISBN 978-5-7996-3257-1. — Текст : непосредственный.

ISBN 978-5-7996-3257-1

На примере языка программирования C++17 вводятся основные концепции структурного программирования. Рассматривается широкий спектр тем: базовые конструкции C++, концепция неопределенного поведения, управление памятью, форматы представления чисел, концепция объекта, си-строки, конечные автоматы, простые структуры данных и алгоритмы сортировки, введение в вопросы организации процесса разработки программ.

Для студентов бакалавриата, обучающихся по направлениям «Механика и математическое моделирование» и «Прикладная математика».

УДК 004.43(075.8)
ББК 32.973.26-018.1я73

На обложке: рисунок Д. Р. Кувшинова

ISBN 978-5-7996-3257-1

Оглавление

Предисловие

Книга предлагает вводный курс в программирование с использованием C++17 в качестве языка программирования. Объем книги разбит на 18 глав. Можно условно отождествить главы с учебными неделями. Последняя глава содержит дополнительный материал, полное освоение которого на данном этапе не требуется.

Авторы не ставят целью полное описание языка или его стандартной библиотеки. За исключением небольшого введения в последней главе, в книге не представлено обобщенное программирование (шаблоны C++), а объектно-ориентированное программирование дано в объеме, позволяющем ввести управление ресурсами в стиле C++, наследование и виртуальные функции не рассматриваются. Отдельное внимание уделено некоторым теоретическим и практическим аспектам, большинство из которых так или иначе проявляются при программировании независимо от выбранного языка. В частности, это касается вопросов вычислительного характера. Подробно рассмотрены числа с плавающей запятой.

Упражнения помечены знаком У.

Введение

В 1950-е годы происходит взрывное развитие компьютеров — универсальных вычислительных машин. Их универсальность заключается в возможности загрузить в устройство программу, определяя таким образом его поведение. Естественно, программы создавались в той форме, которую принимал компьютер, — в машинном коде. К сожалению, машинный код привязан к конкретной архитектуре компьютера, примитивен и неудобен для человека. Для избавления от необходимости писать программы в машинном коде создаются более удобные для человека «языки высокого уровня» (машинный код — «язык низкого уровня»). Причем задачу перевода программы с языка высокого уровня в машинный код может решать сам компьютер. Для этого требуется написать программу-переводчик (транслятор).

Один из ранних языков высокого уровня — Fortran[1], получивший широкое распространение и актуальный поныне[2], создан под руководством Дж. Бэкуса в компании IBM в 1957 г. Целью Fortran было упрощение решения научно-технических вычислительных задач. Затем появился ряд других широко известных языков высокого уровня, однако сами транслято-

[1] От англ. *formula translation* — «перевод формул».

[2] Текущая версия Fortran 2018 определена международным стандартом ISO/IEC 1539-1:2018. См.: ISO/IEC 1539-1-2018. Information technology. Programming languages. Fortran. Pt. 1: Base language (дата введения 28.11.2018).

ры, базовое программное обеспечение (операционные системы, драйверы устройств и т. п.) продолжали писать в машинных кодах для конкретных семейств компьютеров.

Идея универсального языка высокого уровня, позволяющего решать любые задачи без необходимости иметь дело с машинным кодом, появилась довольно рано. Плодом таких усилий стал проект языка Algol[3], реализованный «в коде» в 1960 г. Сам Algol так и не стал популярным средством разработки программного обеспечения, оставшись, по большому счету, университетским проектом, но послужил фундаментом большого числа новых языков программирования.

Одним из языков, основанных на Algol, стал амбициозный проект универсального языка программирования PL/I[4] компании IBM. В случае IBM развитие универсального языка высокого уровня естественным образом следовало из концепции единой архитектуры машинного кода, реализованной в серии суперкомпьютеров IBM System/360. Устройства данной серии были совместимы с точки зрения машинного кода, но могли иметь разный уровень его аппаратной поддержки: какие-то команды могли выполняться непосредственно, а какие-то — с помощью автоматически загружаемых программ («программная эмуляция»).

На деле PL/I оказался слишком сложным, первые трансляторы (1964 г.) не поддерживали его полностью. Однако это не помешало Массачусетскому технологическому институту и компаниям General Electric и Bell Labs в 1964 г. начать проект многопользовательской операционной системы Multics[5], основная часть которой должна была быть реализована на языке высокого уровня — PL/I. Таким образом, PL/I был использован в качестве языка *системного программирования*. Использование языка высокого уровня теоретически позволяет перенести

[3] От англ. *algorithmic language* — «алгоритмический язык».

[4] От англ. *programming language* — «язык программирования».

[5] От англ. *multiplexed information and computing service* — «мультиплексируемая информационная и вычислительная служба».

написанную на нем операционную систему на любые компьютеры, для которых есть транслятор с этого языка высокого уровня, выполнив сравнительно небольшой набор работ по переписыванию модулей, написанных в машинном коде. Первая версия Multics была представлена в 1969 г.

В начале 1960-х годов в Кембридже разрабатывали свой универсальный язык программирования CPL[6]. CPL был сложным Algol-подобным языком. В процессе написания докторской диссертации М. Ричардсом с целью реализации переносимого между различными семействами компьютеров транслятора CPL была создана упрощенная версия CPL под названием BCPL[7] (представлен в 1967 г.). Концепция переносимости BCPL основывалась на разделении транслятора на две части: транслятор с BCPL на низкоуровневый язык под названием «O-код» и транслятор с O-кода в машинный код конкретного компьютера. После этого для переноса компилятора BCPL на новый компьютер было достаточно написать для него транслятор с O-кода. Далее, можно было использовать BCPL для написания транслятора любого другого языка и сделать его переносимым. Таким образом, BCPL стал не универсальным языком программирования, а специализированным языком системного программирования.

С языком BCPL были знакомы сотрудники Bell Labs К. Томпсон и Д. Ричи, принимавшие также участие в проекте Multics. На основе BCPL в 1969 г. ими был создан язык с более компактным синтаксисом (что было важно в те времена из-за очень небольших объемов доступной оперативной памяти), названный B. Данный язык они начали использовать для реализации собственного проекта переносимой операционной системы, названной Unics. Спустя небольшое время проект был «перезапущен»: язык B заменили его развитием — языком C (1972 г.). Название Unics превратилось в Unix. Дальнейший

[6] От англ. *combined programming language* — «комбинированный язык программирования».

[7] От англ. *basic CPL* — «базовый CPL».

успех Unix сделал используемый в ней язык C основным языком системного программирования в мире. Последняя на момент написания книги версия C представлена в 2018 г.[8]

Развитие компьютерной техники позволило ввести в арсенал доступных исследовательских средств *имитационное моделирование*[9] — использование математических моделей для прямого моделирования, т. е. исследования поведения изучаемого объекта на основе поведения модели при заданном сценарии развития. В качестве средства реализации имитационного моделирования О.-Й. Даль и К. Нюгард из Норвежского вычислительного центра разработали основанный на Algol язык Simula (1962 г.). Версия Simula 67 использовала такие понятия, как «объект», «класс», «подкласс», «наследование», «виртуальная процедура», «сборщик мусора», которые позднее стали широко использоваться в последующих языках программирования.

Б. Строуструп использовал Simula 67 для моделирования компьютерных сетей, однако возникла проблема с низкой производительностью интерпретатора Simula, в связи с чем было принято решение переписать программу моделирования на языке C. В результате им был создан набор расширений языка C под названием C with classes («C с классами», 1979 г.). Транслятор C with classes, названный cfront, переводил код в чистый C, который затем транслировался в машинный код. В 1983 г. развитие C with classes получило название C++. Данный подход затем применялся и при разработке других языков программирования: достаточно написать транслятор с нового языка на язык C, и вы можете использовать свой язык на всех компьютерах, для которых есть транслятор C.

Впоследствии C++ стал самостоятельным языком программирования. Более того, ряд современных трансляторов, используемых для трансляции кода на языке C, являются транс-

[8] См.: ISO/IEC 9899-2018. Information technology. Programming languages. C (дата введения 05.07.2018).

[9] В англ. языке используется термин *simulation*.

ляторами C++, поддерживающими режим «чистого C». Общее подмножество C и C++ составляет бóльшую часть стандартного C, из-за чего данные языки иногда не разделяют и обозначают как «C/C++»[10]. И все же область системного программирования в основном принадлежит C, в то время как C++ стал широко используемым при решении прикладных задач универсальным языком программирования с корнями в системном программировании, который также популярен для решения вычислительных научно-технических задач.

Начиная с 1998 г. было выпущено шесть версий стандартного C++. Текущей на момент написания книги версией является C++20[11], однако ее поддержка компиляторами пока не полна.

Значительную часть стандартов C и C++ занимает описание *стандартной библиотеки* — определений, которые не являются частью самого языка, но должны быть предоставлены любой, поддерживающей стандарт, реализацией транслятора. В частности, операции ввода-вывода не входят в состав языка, а предоставляются стандартной библиотекой. Существует также множество иных библиотек, не входящих в стандарт, предоставляющих средства для работы с сетью, графикой, звуком, базами данных и т. д.

C++ предоставляет развитые средства абстракции, предлагая возможность программировать в процедурном, объектно-ориентированном и функциональном стилях и в то же время в случае необходимости (а без действительной необходимости это делать не стоит) приближаться к уровню машинного кода. Изучение C++ позволяет охватить различные аспекты программирования наиболее широко.

---

[10] Среди программистов на языке C использование такого обозначения часто встречает сильное неприятие.

[11] См.: ISO/IEC 14882-2020. Information technology. Programming languages. C++ (дата введения 01.12.2020).

Глава 1

Элементарные понятия

Множество [*set*] — *некая совокупность* **элементов** [*element*].

Предложение выше, хотя и оформлено как определение, на самом деле таковым не является. Слова *множество*, *совокупность*, *набор* можно считать синонимами. Мы не будем здесь давать теорию множеств, ограничившись лишь некоторыми определениями и полагая, что для наших целей достаточно интуитивного понимания основ элементарной теории множеств.

Запись $a \in A$ читается «(элемент) a принадлежит (множеству) A».

Запись $A \subseteq B$ читается «(множество) A является подмножеством (множества) B», т. е. каждый элемент A является и элементом B. Если $A \subseteq B$ и $B \subseteq A$, то эти два множества совпадают ($A = B$).

Запись $A \subset B$ читается «(множество) A является собственным подмножеством (множества) B», т. е. $A \subseteq B$ и известно, что не все элементы B принадлежат A, а значит, эти два множества не совпадают.

Пустое множество [*empty set*] — множество, не содержащее ни одного элемента и являющееся подмножеством любого другого множества. Обозначается $\varnothing$.

Конечное множество [*finite set*] — множество A, для которого найдется $n \in \mathbb{N}$, что все элементы этого множества можно представить в виде последовательности $A = \{ a_1, a_2, \ldots, a_n \}$, т. е. *взаимно-однозначно* отождествить с конечным натуральным рядом $1, 2, \ldots, n$. В этом случае число n называют **размером** множества A: $|A| = n$.

Аксиома: множество $\varnothing$ также причисляется к конечным, при этом $|\varnothing| = 0$.

Бесконечное множество [*infinite set*] — множество, для которого найдется взаимно-однозначное соответствие между им самим и каким-либо его собственным подмножеством.

Например, множество натуральных чисел $\mathbb{N}$ бесконечно, поскольку (с помощью умножения и деления на два) можно установить взаимно-однозначное соответствие между $\mathbb{N}$ и множеством четных натуральных чисел, которое, очевидно, является собственным подмножеством $\mathbb{N}$.

Счетное множество [*countable set*] — бесконечное множество, для которого можно построить взаимно-однозначное соответствие с множеством $\mathbb{N}$. Иными словами, элементы счетного множества можно *пронумеровать* натуральными числами.

Примеры счетных множеств: натуральные числа, четные числа, целые числа, простые числа, пары целых чисел.

Несчетное множество [*uncountable set*] — бесконечное множество, не являющееся счетным.

Мощность множества [*set cardinality*] — обобщение понятия «размер множества» на все множества. Для конечных множеств — то же, что и размер. Для бесконечных множеств используются специальные обозначения. Например, мощность любого счетного множества обозначается $\aleph_0$ («а́леф-нуль»).

Множества, между которыми есть взаимно-однозначное соответствие, называются **равномощными**, т. е. имеющими одинаковую мощность.

Символ[12] [*symbol*] — абстрактная сущность, характеризующаяся тем свойством, что каждый определенный символ равен только самому себе и не равен прочим символам.

Алфавит [*alphabet*] — непустое конечное множество символов.

Так как любой алфавит состоит из конечного числа символов, то вполне естественно будет пронумеровать их и отождествить каждый символ с его номером (натуральным числом) — **кодом** [*code*] символа. Таблица, задающая такое соответствие, называется **кодировкой** [*character set*].

Как можно видеть, английский термин вместо слова *symbol* содержит слово *character*[13], которое может быть переведено как «символ, знак» или «буква», т. е. графический элемент представления текста. Таким образом, *symbol* есть абстрактное (математическое) понятие, а *character* — «технологическое» понятие. В контексте данной книги мы не будем различать эти понятия, используя слово «символ».

Наиболее распространенная кодировка — ASCII[14] — состоит из 128 символов, пронумерованных числами от 0 до 127. Ряд других популярных кодировок базируется на ASCII, добавляя к ней символы с кодами от 128 и выше.

Здесь надо заметить, что под кодировкой обычно понимается не только нумерация некоторого набора символов, но и способ представления этого кода в виде последовательности *бит*. Сам процесс порождения этого представления по символу (или его номеру) называется **кодированием** [*encoding*]. Обратный процесс — **декодированием** [*decoding*]. Вначале для простоты можно полагать, что один символ представим одним байтом, что справедливо для таких кодировок, как: ASCII, Latin-1,

---

[12] От др.-греч. *σύμβολον* — «помета, знак», *συν* — «с, вместе», *βάλλω* — «бросаю, кладу».

[13] От др.-греч. *χαρακτήρ* — «резец», затем также «отпечаток, представление, буква», *χαράσσω* — «вырезаю».

[14] Англ. *American Standard Code for Information Interchange* — «американский стандартный код обмена информацией».

Windows-1251, КОИ-8 (три последних базируются на ASCII, доопределяя еще 128 символов).

Строка [*string*] — конечная последовательность символов, выбранных из некоторого алфавита.

Пустая строка [*empty string*] — строка, не содержащая ни одного символа. Традиционно обозначается ϵ.

Множество всех строк алфавита $\mathcal{A}$ обозначается $\mathcal{A}^*$.

Каков бы ни был алфавит $\mathcal{A}$, всегда истинно утверждение «ϵ принадлежит множеству строк $\mathcal{A}^*$». Это предложение можно записать с помощью квантора всеобщности $\forall$ и связки импликации $\Rightarrow$ как «истинна логическая формула:

$$(\forall \mathcal{A})\ \mathcal{A} \text{ — алфавит} \Rightarrow \epsilon \in \mathcal{A}^*\text{».}$$

Множество $\mathcal{A}^*$ *изоморфно* множеству натуральных чисел $\mathbb{N}$ (с нулем). «Изоморфно» означает «существует *изоморфизм*», т. е. взаимно-однозначное соответствие между элементами этих двух множеств. Чтобы построить изоморфизм, нужно указать способ сопоставления элементу первого множества элемента второго множества и наоборот. В данном случае это сделать достаточно легко.

Представим, что $\mathcal{A} = \{\, a \,\}$ (состоит из единственного элемента). Тогда каждая строка характеризуется лишь своей длиной. Итак, если нам дана строка $\alpha \in \mathcal{A}^*$, то соответствующее число $n = |\alpha|$ (длина строки). Наоборот, если нам дано число $n \in \mathbb{N}$, то соответствующая строка может быть получена повторением n раз символа a. Положим $|\epsilon| = 0$ по определению.

Пусть теперь $|\mathcal{A}| = r > 1$. Тогда к любой строке $\alpha \in \mathcal{A}^*$ мы можем дописать один символ r способами и получить новую строку из $\mathcal{A}^*$ длины $(|\alpha| + 1)$. Таким образом, имеем: строк длины 0 — 1 (пустая строка), строк длины 1 — $1 \cdot r = r$, строк длины 2 — $r \cdot r = r^2$, ..., строк длины k — r^k. Всего строк длины меньше k имеется ровно[15]

$$n_r^k \triangleq 1 + r + r^2 + \ldots + r^{k-1} = \frac{r^k - 1}{r - 1}.$$

[15] Знак $\triangleq$ читается как «по определению равно».

Зададим произвольный порядок на алфавите $\mathcal{A}$, пронумеровав его элементы числами 0, 1, ..., $(r-1)$, и назовем теперь символы $\mathcal{A}$ **цифрами**[16] [*digits*[17]]. Одна цифра *меньше* другой, если соответствующие им числа находятся в том же отношении.

Зададим теперь порядок на строках $\mathcal{A}^*$. Пусть даны две строки $\alpha \in \mathcal{A}^*$ и $\beta \in \mathcal{A}^*$. Теперь, если $|\alpha| < |\beta|$, то считаем, что $\alpha < \beta$. Если $|\alpha| = |\beta|$, то сравним цифры в соответствующих позициях. Если они посимвольно совпадают, то считаем строки равными. Если же есть несовпадения, то эти строки можно представить как составленные из трех частей:

$$\alpha = \gamma a \zeta,$$
$$\beta = \gamma b \eta,$$

где $\gamma \in \mathcal{A}^*$, $|\gamma| < |\alpha|$ — совпадающий *префикс*, $a \neq b$ — первые слева несовпадающие цифры, $\zeta \in \mathcal{A}^*$, $\eta \in \mathcal{A}^*$, $|\zeta| = |\eta| = |\alpha| - |\gamma| - 1 \geqslant 0$ — некие *суффиксы*. Если $a < b$, то и $\alpha < \beta$.

Поставим в соответствие каждой строке количество строк, меньших ее. Так как нет строк, меньших ϵ, то ей соответствует число 0. Построенное отношение строк и натуральных чисел является взаимно-однозначным. Итак, $\mathcal{A}^*$ является счетным множеством.

У Пусть дан алфавит $\{a, b, c\}$ (соответственно, 0, 1, 2). Вычислить число, соответствующее строке *baac*. Вычислить строку, соответствующую числу 100.

Наличие для любого алфавита $\mathcal{A}$ изоморфизма $\mathcal{A}^* \leftrightarrow \mathbb{N}$ дает возможность также определить изоморфизм $\mathcal{A}^* \leftrightarrow \mathcal{B}^*$ для любого алфавита $\mathcal{B}$. Все алфавиты в этом смысле эквивалентны.

Таким образом, можно представлять любые строки (и натуральные числа), используя один удобный алфавит. В случае

[16] От араб. корня *ṣ-f-r* — «быть пустым». Калька с санскр. *śūnya* «пустой» в качестве названия нуля.

[17] От лат. *digitus* — «палец». Англ. *digit* также означает «разряд» (позиция цифры в записи числа).

вычислительной техники таким алфавитом стало множество из двух элементов, обозначаемых 0 и 1. **Бит** [*bit*[18]] — символ в строке, составленной из 0 и 1, двоичная цифра.

У Пусть $|\mathcal{A}| = r > 1$ и $|\mathcal{B}| = \rho > r$. Скольких символов алфавита $\mathcal{A}$ достаточно для того, чтобы, используя описанный выше изоморфизм $\mathcal{A}^* \leftrightarrow \mathcal{B}^*$, представить любую строку из $\mathcal{B}^*$ длины не более n?

Автомат[19] [*automaton*] — абстрактное устройство, реализующее отображение $\mathcal{X}^* \mapsto \mathcal{Y}^*$, где $\mathcal{X}$ — алфавит ввода, а $\mathcal{Y}$ — алфавит вывода.

Данное [*datum*[20]] — любая строка, задействованная в выполнении какой-либо операции. Например, конкретный вход автомата.

Алгоритм [*algorithm*[21]] — полное описание порядка действий, позволяющее некоторому исполнителю за конечное число действий получить требуемый результат.

Автомат можно считать объединением исполнителя и алгоритма, описывающего способ выполнения некоторого преобразования строк.

Алгоритмизация — разработка алгоритма для решения определенной задачи.

Формальный язык [*formal language*] — множество строк над некоторым алфавитом. Таким образом, если $\mathcal{A}$ — алфавит, то любое $L \subseteq \mathcal{A}^*$ является формальным языком над $\mathcal{A}$.

---

[18] От англ. *binary digit* — «двоичный разряд», также *bit* «кусочек» от *to bite* «кусать».

[19] От др.-греч. *αὐτόματον* — «самодвижущееся (устройство)», *αὐτός* — «сам», праиндоевроп. **mn̥t-* — «воля, мысль» (ср. англ. *mind*).

[20] Лат. слово, причастие глагола *do* «даю». Во мн. ч. *data*.

[21] От *Algorizmus* — латинизированного варианта части имени (нисбы) Мухаммада аль-Хорезми, описавшего в своей «Книге об индийском счете» алгоритмы выполнения арифметических действий с числами в десятичной позиционной системе счисления; *-th-* под влиянием др.-греч. *ἀριθμός* — «число» (откуда также *арифметика*).

Язык программирования [*programming language*] — формальный язык, строки которого (**программы** [*programs*[22]]) описывают некие автоматы.

Компьютер[23] — универсальный автомат: устройство, позволяющее моделировать работу автоматов из некоторого *достаточно широкого* класса. Конкретный автомат определяется программой, загруженной в компьютер.

Итак, программу можно считать конкретной реализацией некоторого алгоритма. Один алгоритм можно реализовать в виде разных программ, отвечающих разным дополнительным практическим требованиям или ограничениям реального компьютера (например, по размеру доступной памяти). Реальные программы могут сужать множество доступных к решению задач относительно исходного алгоритма.

Программирование — составление программы, решающей заданную задачу. Алгоритмизация нередко подразумевается частью процесса программирования.

Чем сложнее язык программирования, тем сложнее должен быть принимающий его компьютер. Сложность может выражаться как в сложности *грамматики* (набора правил, определяющих принадлежность строки языку), так и в сложности *семантики* (набора операций или состояний компьютера, отвечающих конструкциям языка).

Язык программирования низкого уровня [*low-level programming language*] — язык программирования, либо принимаемый некоторым компьютером непосредственно (**машинный код**, последовательность чисел — *кодов команд*), либо взаимнооднозначно соответствующий машинному коду (**язык ассемблера**[24]).

---

[22] От др.-греч. *πρόγραμμα* — «объявление, предписание, указ», *προ-* — «пред-» и *γράφω* — «пишу».

[23] Англ. неологизм от лат. *computo* — «считаю».

[24] От англ. *assembler* — «сборщик». Так стали называть программы, формирующие по удобной для человека текстовой записи про-

Языки, не являющиеся языками низкого уровня, называют **языками высокого уровня** [*high-level programming languages, HLL*].

Граница между уровнями расплывчата и определяется в первую очередь практическими и технологическими соображениями: теоретически можно сделать компьютер, непосредственно воспринимающий, например, язык C++, но это сложно и неоправданно с практической точки зрения.

Ряд языков высокого уровня, например: C, BCPL, PL-M и Forth, в каком-то смысле близки к машинному коду целевых компьютеров, поэтому, хотя они и являются языками высокого уровня по данному выше определению, их нередко считают языками низкого уровня или «языками среднего уровня», так как велик контраст по сравнению с языками еще более высокого уровня (есть даже понятие *язык сверхвысокого уровня* [*VHLL*], применяемое к языкам, семантика которых опирается на *виртуальную машину*).

Слово «средний» отражает не только положение между низким и высоким, но и роль данных языков как языков, на которых пишут (или писали) операционные системы, драйверы, системные утилиты и различное *промежуточное программное обеспечение* [*middleware*] — компоненты, используемые другими программами и выступающие «средним звеном» между компьютерной системой и **приложением** [*application*], т. е. программой, решающей некую прикладную задачу. Более удачным представляется термин *языки системного программирования*, хотя он включает в себя больше языков (например, C++).

грамму в машинном коде. Соответственно, программу, выполняющую обратное преобразование, называют *дизассемблер*.

Транслятор — автомат (программа), переводящий программы с одного языка на другой. Язык ввода называется **исходным** [*source*[25]], а язык вывода — **целевым** [*target, object*[26]].
Интерпретатор [*interpreter*] — транслятор, немедленно исполняющий программу-вывод. Таким образом, интерпретатор моделирует компьютер, непосредственно принимающий программу на исходном языке интерпретатора.
Файл [*file*[27]] — имеющая идентификатор единица данных, существующая в рамках некоторой *файловой системы*. Содержимое файла можно считать строкой (даже если файл не является «текстовым»), а *формат файла* — формальным языком.

Файлы специального формата, содержащие целевой код, называют **объектными файлами** или, кратко, *объектами*[28].
Исполняемый файл [*executable file*] — файл специального формата[29], хранящий программу, готовую для немедленного исполнения (обычно в машинном коде).

Операционная система (ОС) сама является набором программ, и для решения пользовательских и вспомогательных задач и управления ими необходима общая концепция выполняемой в данный момент программы.
Процесс [*process*[30]] (также **задача** [*task*]) — программа, выполнение которой управляется операционной системой.

При запуске исполняемого файла ОС загружает его в память, создает новый процесс и запускает его (передает ему часть процессорного времени). В случае наличия технической

---

[25] Соответственно, на жаргоне текст программы на исходном языке называется «исходник» или «source».

[26] От англ. *objective* — «задача, цель». Англоязычный термин, возможно, пошел от целевого кода компилятора BCPL («O-code»).

[27] От лат. *filum* — «нить, волокно». В англ. получило смысл «подшивка», «папка с бумагами, досье, скоросшиватель», «картотека».

[28] Это понятие никак не связано с понятием *объекта* в *объектно-ориентированном программировании*.

[29] Задается операционной системой.

[30] От лат. *processus* — «продвижение, шествие, ход», *procedo* — «продвигаюсь».

возможности процессы могут выполняться одновременно (*параллельно*) в рамках одной системы.

Библиотека [*library*] — набор программ, предназначенный для использования их как части других программ. Обычно конкретная библиотека решает четко очерченный круг задач, хотя бывают и «универсальные библиотеки». Кроме того, современные языки программирования обычно предполагают наличие стандартных библиотек широкого профиля.

Библиотеки могут предоставляться либо в виде исходных текстов на языках высокого уровня, либо в виде специальных файлов — **двоичных библиотек** (библиотек в машинном коде). Последние могут быть двух типов: **статически связываемые** [*statically linked*] и **динамически связываемые** [*dynamically linked, DLL*[31]].

Статически связываемая библиотека должна быть включена в состав исполняемого файла при его создании. Динамически связываемая библиотека может быть загружена и подключена процессом во время его исполнения.

Компилятор [*compiler*] — транслятор, формирующий файл на целевом языке, являющийся объектным файлом, файлом (двоичной) библиотеки или исполняемым файлом.

Концепция компилятора влечет введение понятий **времени компиляции** [*compile time*] и **времени исполнения** [*run time*].

Время исполнения есть собственно период исполнения программы на целевом языке.

Время компиляции отвечает периоду работы компилятора, который помимо собственно перевода программы с исходного языка на целевой может *выполнить* те ее части, что вычисляют значения, не зависящие ни от каких внешних факторов, т. е. константы. Если константа может быть вычислена компилятором заранее, то она называется **константой времени компиляции**.

---

[31] В Unix-подобных ОС используется название *shared object* (*so*).

Компоновщик [*linker*[32]] — приложение, формирующее исполняемый файл из набора объектных файлов (результатов компиляции) и статически связываемых библиотек.

Как было указано выше, некоторые компиляторы могут исполнять функции компоновщика.

Отладчик [*debugger*[33]] — приложение, используемое для **отладки** — поиска ошибок в программе в процессе ее исполнения.

Среда разработки [*development environment*] — программный комплекс, включающий в себя редактор файлов исходного кода, транслятор (и, если требуется, компоновщик) и отладчик.

Интегрированная среда разработки [*IDE*] — среда разработки, включающая в себя «основное приложение» на основе редактора файлов исходного кода, позволяющая вызывать транслятор и задействовать отладчик, не покидая основное приложение.

Как правило, интегрированные среды разработки оперируют таким понятием, как *проект*.

Проект [*project*] — организованный набор файлов (исходный код, настройки, библиотеки и т. д.), обычно размещаемый в одной локации и содержащий данные, позволяющие среде разработки создать приложение или двоичную библиотеку. Эта процедура создания называется **сборка проекта** [*build*].

Таким образом, работа в IDE начинается с создания проекта. В контексте данной книги нам в качестве исходного проекта всегда будет нужен **пустой проект консольного приложения**.

---

[32] От англ. *to link* — «связывать». Компоновщик также называют *редактор связей*.

[33] В конечном счете от англ. *bug* — «жук или любое насекомое», которое на жаргоне стало означать ошибку в коде.

Консольное приложение [*console*[34] *application*] — приложение, работающее в режиме текстового ввода-вывода, реализованное с помощью стандартного интерфейса, предоставляемого ОС (называемого *консоль* или *терминал*[35]).
Единица трансляции [*translation unit*] — файл исходного кода, из которого компилятором порождается отдельный объектный файл. В случае C++ такие файлы обычно имеют расширение cpp, cxx или cc. Некоторые IDE считают единицей трансляции любой файл с таким расширением, включенный в проект.
Заголовочный файл [*header file*] или просто «заголовок» — файл исходного кода, не являющийся единицей трансляции. Данный термин характерен для языков C (расширение файла h) и C++ (расширения файла hpp, hxx, hh или h), в которых во время компиляции текст заголовков попросту вставляется в использующие их единицы трансляции.

Слово «заголовок» возникло вследствие того, что основным назначением заголовочного файла в языке C было связывание разных единиц трансляции или статически связываемых библиотек, поэтому текст заголовков содержал только *объявления* функций и переменных, в то время как их *определения* находились в каких-то единицах трансляции или библиотеках. Это все равно что взять толковый словарь, выписать из него только слова (заголовки словарных статей) и сказать «определения этих слов ищи в прилагающихся книгах». Поиском определений в объектных файлах занимается компоновщик.

Для упрощения разбора текста программы работа транслятора может разделяться на два этапа: *лексический анализ* и *синтаксический анализ*. Лексический анализ заключается в преобразовании исходной последовательности символов в последовательность *лексем* — записей констант, имен (идентифи-

[34] Изначально строительный термин *консоль* был перенесен на рабочее место оператора ЭВМ. Впоследствии стал обозначать средство управления системой на основе текстового ввода-вывода.

[35] От лат. *terminus* — «граница», т. е. изначально «пограничное» устройство между (большим) компьютером и пользователем.

каторов), знаков пунктуации, операций, ключевых слов и т. п. Синтаксический анализ заключается в выявлении собранных из лексем конструкций языка.

Авторы не ставят своей целью дать полное, подробное или формальное описание синтаксиса C++, однако в конце главы приведены таблицы ключевых слов, пунктуации и операций.

Ключевые слова можно считать своеобразной пунктуацией, их нельзя использовать в качестве идентификаторов (кроме контекстных ключевых слов). Зарезервированные слова C++17 были ключевыми словами в предыдущих стандартах языка, их нельзя использовать.

В некоторых случаях смысл символов зависит от контекста: например, знаки <, > и , могут выступать в роли операций или в роли пунктуации. Некоторые ключевые слова в C++ также относят к операциям, например, sizeof, alignof, static_cast. Некоторые из них являются альтернативными обозначениями операций.

Задания для самопроверки

У Определите истинность следующих высказываний:

- Транслятор — это автомат.
- Транслятор — это программа.
- Интерпретатор — это автомат.
- Компилятор — это интерпретатор.
- Компилятор выполняет трансляцию машинного кода в текст на языке высокого уровня.
- Заголовочный файл транслируется в отдельную единицу трансляции.
- Доступ к библиотекам обязательно осуществляется с помощью заголовочных файлов.

- Файл — это строка.
- Алфавит может быть пустым.
- Формальный язык может быть пустым.
- Суффикс — это средняя часть строки.
- 7 бит достаточно для представления чисел 0, 1, ..., 100.
- Любое множество является либо конечным, либо счетным, либо несчетным.
- Два конечных множества, содержащие одинаковое количество элементов, изоморфны.
- Интегрированная среда разработки может содержать интерпретатор.
- Компилятор может быть также интерпретатором.
- Транслятор может быть библиотекой.
- Компоновщик компонует заголовочные файлы в исполняемый файл.
- Отладчик — это человек, который занимается отладкой.
- Зарезервированные слова — это лексемы.
- Слово repeat является ключевым в C++17.
- В C++ имеется операция, обозначаемая ; (точкой с запятой).

Список ключевых слов C++17

alignas	dynamic_cast	short
alignof	else	signed
and	enum	sizeof
and_eq	explicit	static
asm	extern	static_assert
auto	false	static_cast
bitand	float	struct
bitor	for	switch
bool	friend	template
break	goto	this
case	if	thread_local
catch	inline	throw
char	int	true
char16_t	long	try
char32_t	mutable	typedef
class	namespace	typeid
compl	new	typename
const	noexcept	union
constexpr	not	unsigned
const_cast	not_eq	using
continue	nullptr	virtual
decltype	operator	void
default	or	volatile
delete	or_eq	wchar_t
do	reinterpret_cast	while
double	return	xor
		xor_eq

Контекстные ключевые слова C++17

final override

Зарезервированные слова C++17

export register

Пунктуация C++17

( ) [ ] { } < > , ; :

Операции C++17

+	−	*	/	%	!	&&	||
++	−−	&	|	^	~	<<	>>
<	>	<=	>=	!=	==	?:	,
.	−>	.*	−>*	=	+=	−=	*=
	/=	%=	&=	|=	^=	<<=	>>=

Альтернативные обозначения операций C++17

and	&&	and_eq	&=	bitand	&
or	||	or_eq	|=	bitor	|
not	!	not_eq	!=	compl	~
xor	^	xor_eq	^=		

Глава 2

Значения и переменные

Будем называть **конкретной сущностью** всякий объект, мыслимый как часть физического реального (или воображаемого) мира. Конкретные сущности имеют некое место во времени и пространстве. Как правило, конкретные сущности слишком сложны, чтобы их можно было *описать точно*.

Описание конкретной сущности требует *абстракции*[36] — отделения существенных черт сущности от несущественных. О несущественных чертах мы затем забываем. Полученный набор существенных черт (*свойств*) составляет *модель*[37] — некий конструкт, которым мы подменяем исходную сущность в своих рассуждениях, программах или опытах, перенося затем полученные результаты на исходную сущность.

В программах или математических выкладках мы оперируем моделями, заданными, в конечном итоге, строками. Каждое свойство представлено некоторой строкой, принадлежащей некоему формальному языку.

Конечно, человек может оперировать материальными объектами в качестве моделей или образами в своем воображении.

---

[36] От лат. *abstractio* — «отделение», *ab* — «от, прочь», *traho* — «тащу, влеку».

[37] Народнолат. вариант лат. *modulus* — «мерка» (откуда также слово *модуль*), от *modus* — «мера, предел, способ».

Чем это отличается от строк формальных языков? Материальные объекты сами являются конкретными сущностями, а значит, каждая физическая модель уникальна и сложна как конкретная сущность. Образы могут быть субъективны и привязаны к конкретному человеку. В то же время математические объекты, к коим принадлежат формальные языки, заданы точно и доступны для интерпретации другими людьми или компьютерами.

В отличие от конкретных сущностей, математические объекты (**абстрактные сущности**) задаются введением[38], а не отнятием свойств. И свойства эти так или иначе сводятся к множествам строк. Они происходят из мира идей[39] и понятий, выстроенных людьми. Но математические объекты отличаются от прочих идей максимальной степенью десубъективизации. Наверное, не будет ошибочным заявить, что любая идея, сформулированная точно, становится математическим объектом.

Значение [*value*] (свойства) — абстрактная сущность, сопоставленная свойству.

Для выполнения операций над значениями требуется их как-то представлять. Мы можем считать, что все действия выполняются над строками, поэтому в качестве **представлений** [*representations*] значений мы будем использовать строки[40].

Операция[41] [*operation*] — отображение, ставящее в соответствие набору **операндов** новое значение — **результат операции**.

---

[38] «Аксиоматический метод».

[39] Слово *идея* происходит от др.-греч. *εἰδέα* — «вид, форма», праиндоевроп. **weyd-* — «видеть».

[40] Например, последовательность бит в памяти — строка над алфавитом $\{0, 1\}$, где 0 и 1 — условные обозначения двух различимых состояний бита.

[41] От лат. *operatio* — «действие», *operor* — «действую, работаю», *opus* — «работа, деятельность».

В зависимости от числа операндов говорят об **унарной** (один операнд), **бинарной** (два операнда), **тернарной** (три операнда) операции.

Особенность понятия *операции* носит, скорее, культурно-денотативный характер. Операциями принято называть ряд арифметических и логических действий, а также их всевозможные обобщения, записываемые с помощью того или иного знака, обычно размещаемого между операндами (*инфиксная форма записи*).

Тип[42] [*type*] — множество значений, для которого, как правило, определен также некоторый набор операций. С практической точки зрения можно сказать, что «тип» — это метка, которая прикрепляется транслятором к представлениям значений, чтобы определять, что это такое и как с этим можно работать, какой действительный смысл имеют те или иные операции.

Простейшая программа на C++, которая ничего не делает, имеет следующий вид:

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

Рассмотрим общую структуру программы на C++. Текст на C++ состоит из:

- комментариев (игнорируемых компилятором);
- директив препроцессора, выполняемых до компиляции;
- директив компилятора;
- объявлений;
- определений.

[42] От лат. *typus*, др.-греч. *τύπος* — «оттиск, отметина», *τύπτω* — «ударяю».

Комментарий [*comment*] — текст, оставленный программистом для читателя-человека[43]. Имеет специальное оформление, что позволяет компилятору пропускать этот текст.

В C++ есть два вида комментариев: *однострочные* и *многострочные*. Их оформление поясним на примере.

Пример 2.1. HelloWorld

```
// Однострочный комментарий.
/* Многострочный
   комментарий. */
/* Чтобы иметь возможность использовать
   стандартные средства работы с консолью,
   следует подключить iostream. */
#include <iostream>
// Отсюда начинается выполнение программы.
int main() {
  // Между и находятся инструкции,
  // выполняемые функцией.
  // Вывести строчку Hello, world!:
  std::cout << "Hello, world!";
}
```

Последовательность символов */ всегда закрывает многострочный комментарий, независимо от того, сколько было до этого открывающих последовательностей /*.

C++ предлагает ряд встроенных типов, среди которых перечислим пока следующие четыре:

- **int**[44] — целое число, представление которого имеет ограниченную ширину в битах (обычно 32). Константы записываются следующим образом:
  - нуль: 0;

[43] На деле комментарии нередко могут содержать директивы для внешних инструментов, обрабатывающих программу. В частности, генерирующих документацию (пример: Doxygen).

[44] От англ. *integer* — «целое (число)», лат. *integer* букв. «нетронутый», откуда далее значения «неповрежденный, целый, полный».

- в десятичной системе, если начинаются с 1–9: 114, 1'000 (допустимо использовать апостроф ' как разделитель разрядов);
- в двоичной системе, если начинаются с 0b или 0B: 0b1110'0010;
- в восьмеричной системе, если начинаются с 0, за которым идет цифра: 077 равно не 77_{10}, а 77_8, т. е. 63_{10};
- в шестнадцатеричной системе, если начинаются с 0x или 0X: 0xABCD, 0xDead'Beef, буквы A–F используются для обозначения цифр со значениями 10–15, регистр букв на значение числа не влияет.

- **float**[45] — число с плавающей запятой (точкой) «одинарной точности». Подробнее числа с плавающей запятой будут рассмотрены позднее. Пока можно считать, что это некоторое (возможно, неточное) представление десятичной дроби.

  При записи значений данного типа в качестве разделителя целой и дробной частей используется точка, в конце ставится суффикс f: 3.141593f. Допустимо опускать нулевую дробную часть: 0.f, или нулевую целую часть: .25f. После дробной части можно указать показатель степени десятки, на которую домножается число: например, 1.5e10f есть запись для $1.5 \cdot 10^{10}$.

- **double**[46] — число с плавающей запятой «двойной точности». Запись констант данного типа не требует указания суффикса: 1., .0, 0.5, 2e2 — все эти константы имеют тип **double**.

[45] От англ. *floating point* — «плавающая точка».
[46] От англ. *double precision* — «двойная точность».

- **char**[47] — является основным встроенным символьным типом и также считается синонимом *байта*[48] — минимальной отдельно адресуемой ячейки памяти.

  Его размер обычно совпадает с традиционным размером байта в 8 бит, хотя можно столкнуться с системами, в которых **char** имеет другой размер, например, 32 бита. Стандарт языка требует лишь, чтобы представление **char** занимало не меньше, чем 8 бит.

  В программе значения типа **char** записываются путем заключения символа в апострофы: 'a' — буква «a» (кодировку выберет компилятор), символ \ используется для указания специальных символов или прямого указания кода символа:

  – '\0' — символ с кодом 0;
  – '\'' — апостроф ';
  – '\\' — обратная косая черта \;
  – '\n' — символ перевода строки (*line feed*);
  – '\r' — символ возврата каретки (*carriage return*);
  – '\t' — символ табуляции (переход к следующей колонке);
  – '\b' — символ забоя (*backspace*, возврат на один символ влево, но в пределах одной строки);
  – '\a' — символ звукового сигнала (попробуйте вывести его в консоль);
  – '\147' — символ с кодом 147_8 ('g' в кодировке ASCII);
  – '\xff' — символ с кодом FF_{16} (255).

Целочисленные значения автоматически приводятся к типам чисел с плавающей запятой при выполнении операции,

[47] От англ. *character* — «символ».
[48] Англ. *byte* от *to bite* аналогично *bit*.

один операнд которой является целым числом, другой — дробью.

Символы считаются целыми числами (мы работаем с кодами символов) и автоматически приводятся к типу **int** при выполнении с ними арифметических действий.

Запись сразу нескольких символов, например, 'abcd', однако, имеет тип **int** и обычно интерпретируется компилятором как поразрядно совпадающее с этой последовательностью байт представление целого числа.

Приведение типа [*type conversion*] — операция преобразования значения одного типа в значение другого типа, максимально близкое в каком-то смысле к эквиваленту исходного значения. Например, преобразовав целое число 2 в тип **double**, мы получим 2.0, что есть представление того же числа, но в формате **double**. С другой стороны, преобразовав 2.6 в тип **int**, мы получим 2, дробная часть будет отброшена.

Иногда «приведением типа» также называют номинальную *смену типа* [*type casting*[49]], по выполнении которой транслятор просто начинает считать то же самое представление исходного значения представлением значения другого типа.

Приведение типа называется **явным** [*explicit*], если оно явно затребовано в коде программы. В C++ существует несколько форм явного приведения типов, самыми простыми являются следующие две (для простоты их пока можно считать эквивалентными):

(тип)значение
тип(значение)

Например, **int**('a') вернет код символа «a», представленный значением типа **int**.

[49] Часто термин «type cast» применяют и для операции преобразования значения. В случае си-подобных языков так принято называть именно явное приведение типа, независимо от того, является оно номинальным или нет.

Директива [*directive*] — указание в тексте, управляющее работой транслятора. Директивы не порождают целевого кода.

Директивы компилятора будут вводиться в книге по мере необходимости. Одна из часто используемых директив — **using namespace**. Например, после **using namespace** std; в функции или единице трансляции, где расположена эта директива, уже не требуется писать std:: для выборки имен из *пространства имен* std стандартной библиотеки C++.

Препроцессор [*preprocessor*[50]] — программа (вид транслятора), преобразующая исходный код программы до передачи его компилятору. Преобразование заключается в выполнении текстовых замен.

В случае языка C (и по наследству, C++) есть заданный стандартом препроцессор, в настоящее время являющийся частью компилятора. Именно его мы имеем в виду под словом «препроцессор». Директивы препроцессора занимают, по крайней мере, одну строчку и начинаются символом #. Наиболее популярная директива препроцессора — #**include**. Краткое описание директив препроцессора размещено в дополнительном материале.

Определение [*definition*] — конструкция на языке программирования, которая полностью определяет некоторый именованный элемент программы.

Правило одного определения [*One Definition Rule, ODR*]: любой элемент программы может быть определен только один раз.

Объявление [*declaration*] — конструкция на языке программирования, которая объявляет существование некоторого именованного элемента программы («знакомит» компилятор с именем), но не определяет его целиком. Обычно выглядит как усеченное определение (без собственно содержания).

Наличие объявления подразумевает наличие соответствующего определения где-то в другом месте программы. Необходи-

[50] От лат. *pre-* — «пред-», англ. *processor* — «обработчик» (от *process*), т. е. «предобработчик».

мость в объявлениях возникает при разрешении *циклических определений* (когда определение A зависит от определения B, а B, в свою очередь, зависит от A). Поскольку текст программы есть линейная последовательность элементов, просматриваемая компилятором «сверху вниз», то в каждом конкретном месте программы компилятор еще «не знает» те элементы, которые стоят ниже по тексту.

Также объявления используются для разделения программы на *части* (*модули*) и связывания этих частей. То, что определено в одной части программы, объявляется в использующих это определение других частях программы. Таким образом, объявления могут быть использованы для облегчения восприятия программы человеком, поскольку человек не может воспринять большую программу как единое целое — требуется разбивать ее на части и воспринимать по частям.

Любой элемент программы может быть объявлен сколько угодно раз.

Переменная [*variable*] — имя в программе, привязанное к значению. Слово «переменная» подразумевает возможность изменения.

Возможны два подхода к практической реализации переменных: «переменные-ссылки» и «переменные-ящички».

При первом подходе переменная каким-то образом ссылается на некое значение (говорят, что «переменная привязана к значению»). Несколько переменных могут ссылаться на одно и то же значение. Изменение переменной может не затрагивать само значение (его представление в памяти компьютера), а заключаться в изменении привязки переменной.

При втором подходе переменная есть *имя* места в памяти, в котором хранится представление значения переменной. У каждой переменной в этом случае свое собственное представление, и изменение переменной заключается в изменении этого представления. C++ реализует именно второй подход.

Как представления, так и сами имена (переменные) могут быть помечены типом, которому принадлежат рассматривае-

мые значения. Если типом помечены представления, то говорят о *динамической типизации*. Если типом помечены переменные, то говорят о *статической типизации*. C++ использует статическую типизацию.

Определение переменной в C++ имеет вид:

тип имя инициализатор;

Итак, тип надо указывать обязательно. Имя переменной — произвольный свободный в данной области видимости *идентификатор*. Инициализатор нужен для задания начального значения переменной и обычно может отсутствовать. Если инициализатор отсутствует, то это может влечь либо инициализацию неким известным начальным значением по умолчанию, либо отсутствие инициализации и, соответственно, неопределенное значение переменной. Какой вариант реализуется, зависит от типа. В дальнейшем этот вопрос будет рассмотрен подробнее.

Идентификатор [*identifier*] — имя в программе, позволяющее обратиться к той или иной конструкции (тип, переменная, функция и т. д.).

В C++ идентификаторы должны начинаться с буквы или знака подчеркивания _ и могут содержать буквы, цифры и знаки подчеркивания. Заглавные и строчные буквы считаются разными, поэтому, например, PI, pI, Pi и pi суть четыре разных идентификатора. Идентификаторы не могут совпадать с ключевыми словами языка. Не следует определять свои идентификаторы, начинающиеся на два знака подчеркивания (например, __in) или на знак подчеркивания и заглавную букву (например, _Bool), так как они зарезервированы для расширений языка.

Определим переменную, которая будет хранить представление некоторого целого числа:

```
int n;
```

Это определение не содержит инициализатора, и в данном случае начальное значение переменной не определено, что может создавать определенные неудобства, поэтому рекоменду-

ется назначить переменной какое-нибудь начальное значение. Например, нуль:

```
#include <iostream>
int main() {
  int n = 0;
  std::cout << n; // должно вывести 0
}
```

C++ позволяет определить сразу несколько переменных одного типа, перечислив их (вместе с инициализаторами, если они есть) через запятую:

```
int one = 1, two = 2, three = 3;
std::cout << one << two << three; // 123
```

Как видно из примеров, стандартным способом вывода текстовых значений в терминал пользователя является их «отправка» в «cout» с помощью операции <<. Напротив, чтобы считать значение, введенное пользователем, можно воспользоваться «cin» и операцией >>:

```
int n = 0;
std::cout << n; // 0
std::cin >> n;  // терминал ожидает ввода
std::cout << n; // введенное число
```

Имена std::cin и std::cout[51] привязаны к *стандартным потокам ввода-вывода* — объектам операционной системы, позволяющим стандартизовать взаимодействие приложений друг с другом и пользователем (используя текстовую форму представления данных). Формально стандартные потоки ввода-вывода считаются текстовыми файлами.

Помимо cin и cout доступны еще std::cerr и std::clog, привязанные к стандартному потоку вывода сообщений об ошибках. Наличие такого отдельного потока позволяет разделить вывод программой данных и сообщений об ошибках (например, *перенаправляя* их в разные файлы).

[51] Сокращения от англ. *character input/output* — ввод/вывод символов.

Вызов cin.ignore позволяет пропускать символы, что иногда используется при организации ручного ввода:

```
#include <iostream>
int main() {
  float f = 0;
  std::cin >> f;
  std::cout << f*f;   // квадрат f
  std::cin.ignore(); // задержка экрана
}
```

В случае ошибки ввода поток переходит в ошибочное состояние, в котором операции ввода-вывода не выполняются. Сбросить ошибки можно вызовом clear:

```
  int a = 42, b = 23;
  // Попробуйте ввести буквы:
  std::cin >> a >> b;
  std::cout << a << '\n' << b << '\n';
  char next = 0;
  // Без сброса ошибки дальше читать не будет:
  std::cin.clear();
  std::cin >> next;
  std::cout << next;
```

В случае, когда мы пытаемся прочитать значение в переменную из потока, уже находящегося в состоянии ошибки, значение переменной не изменяется. В случае же, когда ошибка происходит во время чтения числа, в переменную записывается максимальное представимое число (если на ввод подано слишком большое число), минимальное представимое число (если на ввод подано отрицательное число, слишком большое по абсолютной величине) и нуль в случае, если число вообще нельзя прочитать.

Оператор [*operator*] (в C++) — операция, определенная стандартом языка и включенная в его синтаксис.

Набор операторов фиксирован. В частности, это арифметические операторы +, − (унарный вариант — отдельный оператор), ∗, / и %. Для симметрии определен также унарный +.

При применении к числу он не меняет его значение (*тождественный оператор*).

Выражение [*expression*] — последовательность операндов (непосредственно заданных значений или идентификаторов) и связывающих их операторов, определяющая способ вычисления некоторого значения.

Итак, в случае успеха вычисления выражения мы получаем некоторое значение, которое называется *значением выражения* [*expression value*]. Кроме того, всякое выражение имеет тип, которому должно принадлежать его значение (*тип (значения) выражения*). Благодаря тому, что в C++ переменные имеют тип, тип выражения может быть определен во время компиляции *без* вычисления его значения.

Для явного управления порядком связывания операндов и операций можно использовать круглые скобки ( ).

Атомарное выражение — непосредственно заданное значение или идентификатор. Атомарное выражение не содержит операторов. Например, 2 и pi — атомарные выражения, а 2*pi и (pi) — уже нет.

Вре́менное значение [*temporary value*] — значение любого[52] неатомарного подвыражения в выражении, получающееся в процессе вычисления значения выражения. В конечном итоге, временным значением будет также являться и значение самого выражения целиком.

Временными данные значения называются по той причине, что они нужны только в процессе вычисления выражения. После того, как значение выражения вычислено и как-то использовано, все временные значения уже не нужны и могут быть «забыты» (их представления могут быть уничтожены).

Например, вычисление значения выражения

---

[52] В данном случае имеется в виду некий безымянный результат операции, поскольку помимо таких новых значений значением неатомарного выражения может выступать *ссылка* на уже существующее значение, не являющееся временным (пример: (pi) — ссылка на переменную pi). Ссылки будут рассмотрены подробнее далее.

```
(k*i + j)*4 + s
```

подразумевает вычисление временных значений (будем обозначать их символом $):

```
$0 = k*i;
$1 = $0 + j;
$2 = $1*4;
$3 = $2 + s;
```

где $3 есть значение всего выражения целиком.

Так как промежуточные временные значения используются только один раз, то компилятор может организовать вычисление, используя лишь одну ячейку памяти для хранения промежуточных значений:

```
$0 ← k*i;
$0 ← $0 + j;
$0 ← $0*4;
$0 ← $0 + s;
```

Побочный эффект [*side effect*] — потенциально видимое извне изменение состояния компьютера, включая изменение значений переменных и любые операции ввода-вывода (взаимодействие с внешним миром).

Например, `std::cout << '\n'` есть выражение, значением которого является *сам* объект `std::cout`, а побочным эффектом вычисления — вывод перевода строки в стандартный поток вывода. Бинарный оператор `<<` называется *оператор сдвига влево*, оператор `>>` — *оператор сдвига вправо.*

Так как `<<` связывается слева направо, то

```
std::cout << a << b
```

есть то же самое, что

```
(std::cout << a) << b
```

Значением выражения `(std::cout << a)` является `std::cout`, поэтому «цепочка» `<<` работает как последовательность операций вывода.

Инструкция [*statement*[53]] — элементарная конструкция языка, заключающая в себе некоторое действие.

Пустая инструкция [*null statement*] — инструкция, не предполагающая выполнение каких-либо действий.

Пустая инструкция может понадобиться там, где правила языка требуют размещения инструкции, но с точки зрения программиста не требуется выполнять никаких действий. Простейшая пустая инструкция:

```
;
```

Инструкция-выражение — инструкция, заключающаяся в реализации побочных эффектов вычисления выражения. Наиболее часто встречающийся вид инструкции в C++ имеет вид:

```
выражение ;
```

Пример — уже встречавшиеся выше инструкции вывода. Так,

```
std::cout << pi*r*r
```

есть выражение, а

```
std::cout << pi*r*r;
```

уже является полноценной инструкцией.

Точка с запятой не является разделителем инструкций, а показывает, что текст инструкции закончен.

Если выражение не несет побочных эффектов, то соответствующая инструкция-выражение является пустой инструкцией и может быть отброшена компилятором:

```
2 + 2; // корректная, но пустая инструкция!
```

Присваивание [*assignment*] — действие изменения значения переменной. В C++ присваивание воплощено в форме бинарного оператора = и, таким образом, может быть частью выражения[54]. Итак, присваивание имеет вид:

---

[53] Обычно переводят как «оператор», но в контексте C++ требуется различать понятия *statement* и *operator*.

[54] Во многих других языках программирования (например, Pascal и Python) присваивание реализовано в виде особой инструкции и не может быть частью выражения.

переменная = новое-значение

и вычисляется справа налево, что позволяет записать цепочку присваиваний:

```
a = b = 0
```

есть то же, что

```
a = (b = 0)
```

и записывает 0 в b, а затем записывает значение b (т. е. 0) в a.

Формально изменение значения переменной является побочным эффектом вычисления выражения, содержащего присваивание. Слово «побочный» не должно сбивать с толку: ради этого эффекта присваивание и записывается. То же касается ввода-вывода и других источников побочных эффектов. Если побочный эффект возникает не по осознанной воле программиста, то это, скорее всего, ошибка.

Типом выражения (a = b) можно считать тип переменной a, значением — *саму переменную* a (точнее, ссылку на a).

В C++ доступно также *составное присваивание*, позволяющее писать конструкции вида

```
a = a ∘ b
```

где вместо ∘ можно подставить ряд бинарных операторов (в частности, +, −, ∗, /, %, << и >>), как

```
a ∘= b
```

Составное присваивание можно считать просто сокращением записи, но семантически оно воплощает особое действие преобразования значения переменной, например, «добавить» это +=, «домножить» — ∗= и т. д.:

```
x *= 2; // удвоить x
```

Так же, как и обычное присваивание, составное присваивание вычисляется справа налево.

Запись ++i (унарный оператор ++) называется *инкрементом* i и, как правило, эквивалентна записи i += 1, т. е. вычисление данного выражения увеличивает значение переменной

на единицу, и его значением становится новое значение переменной. Оператор `--` (*декремент*) воплощает собой действие, обратное инкременту.

Помимо «простых» инкремента и декремента в C++ есть также *постинкремент* и *постдекремент*. Они получаются, если операторы `++` и `--` ставить не перед операндом, а после него. Отличие заключается в том, что постинкремент и постдекремент возвращают старое значение переменной:

```
int n = 0;
std::cout << n;    // 0
std::cout << ++n; // 1
std::cout << n++; // 1
std::cout << n;    // 2
std::cout << --n; // 1
std::cout << n--; // 1
std::cout << n;    // 0
```

Задания для самопроверки

У Определите истинность следующих высказываний:

- Строчка

  ```
  std::cout << "Greetings there!\n"
  ```

  является инструкцией.

- Строчка

  ```
  1*2*3*4*5*6;
  ```

  является выражением с синтаксической ошибкой.

- Строчка

  ```
  a + (b = c)
  ```

  является выражением, побочным эффектом которого является запись значения переменной b в переменную c.

- Строчка

  ```
  a = b = c;
  ```

  является инструкцией, содержащей два присваивания.

- Строчка

  ```
  a *= b *= c
  ```

  является выражением, содержащим два присваивания.

- Строчка

  ```
  a *= b *= 2;
  ```

  удваивает значения переменных a и b.

- Вычисление выражения

  ```
  a *= 4*(n - k) + 1
  ```

  подразумевает вычисление четырех временных значений.

У Запишите одной строчкой:

- фрагмент кода:

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

- фрагмент кода:

  ```
  std::cin >> login;
  std::cin >> pwd;
  ```

- фрагмент кода:

  ```
  int x = 0;
  int y = 0;
  ```

- фрагмент кода:

```
x = 10;
y = 10;
z = 10;
```

- фрагмент кода:

```
a *= 3;
b += a;
```

У Чему равны значения переменных после выполнения:

- фрагмента кода:

```
int a, b, c;
a = b = c = 0;
```

- фрагмента кода:

```
int a = 1, b, c;
```

- фрагмента кода:

```
int a = 1, b, c;
b = c = 2;
```

- фрагмента кода:

```
int a, b, c;
a = b = c = 0;
c = a = b = 1;
```

- фрагмента кода:

```
int a, b = 0, c = 0;
a = c = 10;
```

- фрагмента кода:

```
int a = 1, b = 2, c = 3;
a = b = c;
```

- фрагмента кода:

```
int a = 1, b = 2;
int c = a + b;
```

- фрагмента кода:

```
int a, b, c;
a = b = (c = 0) + 1;
```

- фрагмента кода:

```
int a = 1, b = 2, c = a;
a = b; b = c;
```

- фрагмента кода:

```
int a, b, c;
a = b = 1;
c = b = 2;
```

- фрагмента кода:

```
int a, b, c;
a = b = 1;
c = a = 2;
b = c = 3;
```

- фрагмента кода:

```
int a, b = 1;
b = b;
```

- фрагмента кода:

```
int a = 2, b = 3;
a += b;
b *= 2;
```

- фрагмента кода:

```
int a = 5, b = 7;
a += b;
b = a - b;
a -= b;
```

- фрагмента кода:

```
int a = 23, b = 42;
b += a -= b;
a = b - a;
```

Глава 3

Функции

Функция[55] [*function*] — в C++[56] именованный участок кода, который можно *вызывать* [*call*].

Итак, необходимо описать два аспекта функций: определение и вызов.

Определение функции выглядит следующим образом:

```
тип-возвращаемого-значения имя-функции(параметры-функции)
тело-функции
```

Первая строчка называется *заголовок функции*. Если требуется функцию только объявить, то выписывается ее заголовок, после которого ставится точка с запятой:

```
тип-возвращаемого-значения имя-функции(параметры-функции);
```

Объявление функции также называют *прототипом функции*.

Тело функции заключается в фигурные скобки {} и может содержать комментарии, директивы, объявления, определения и инструкции.

---

[55] От лат. *functio* — «исполнение, совершение», *fungor* — «исполняю, совершаю, завершаю».

[56] Отличается от математического понимания функции как отношения между множествами. Включает в себя также понятие *процедуры* или *подпрограммы* из других языков программирования.

Единственный пример функции, который встречался до сих пор в этой книге, — функция main:

```
int main() { /* ... */ }
```

Из данного кода мы можем видеть, что типом возвращаемого значения является **int**, т. е. функция *возвращает* некоторое целое число. Именем является идентификатор main. Список параметров пуст, т. е. функция ничего не принимает. Однако даже в этом случае надо ставить круглые скобки.

Функция main играет особую роль — она является *точкой входа*. Имя main в этой роли закреплено стандартом C++. Данную функцию запрещается вызывать из самой программы (ее вызывает операционная система), а типом возвращаемого значения всегда должен быть **int**.

Точка входа [*entry point*] — функция, вызов которой запускает приложение.

Вызов фунции [*function call*] — стандартная последовательность действий, заключающаяся в переходе исполнителя (процессора) на исполнение тела вызываемой функции и обеспечивающая доступ из данного кода к значениям, переданным в функцию, возврат исполнителя в точку вызова функции после того, как тело функции выполнено, и подстановку в точку вызова возвращенного функцией значения.

Соглашение вызова [*calling convention*] — полный набор правил, определяющих способ выполнения вызова функций на данной программно-аппаратной платформе. Соглашение вызова обычно определяется разработчиком архитектуры команд процессора, но может также (до)определяться операционной системой и компилятором.

Вызов функции в C++ является выражением, которое записывается следующим образом:

```
имя-функции(список-аргументов)
```

Аргумент [*argument*], также *фактический параметр* — значение, передаваемое в функцию в точке вызова.

Параметр [*parameter*], также *формальный параметр* — переменная, инициализируемая аргументом в процессе вызова функции.

И аргументы в месте вызова, и параметры в заголовке функции разделяются запятыми. Так как параметры фактически являются переменными, требуется указывать их тип. В отличие от определения обычных переменных в случае параметров тип требуется указывать для каждого параметра в списке.

В вызове типы аргументов **не** указываются. Компилятор и так знает как типы аргументов (типы соответствующих выражений), так и типы параметров (указаны в объявлении или определении функции). Типы аргументов должны быть *совместимы* с типами соответствующих параметров. Они совместимы, если переменную типа параметра можно инициализировать данным аргументом, т. е. их типы совпадают или определено приведение типа, например, **int** и **float** совместимы в обе стороны:

```
float x = 10; // int 10 → float 10.f
int n = 1.5f; // float 1.5f → int 1
```

При приведении дроби к целочисленному типу дробная часть отбрасывается, и компилятор может (но не обязан) выдавать по этому поводу предупреждение.

Выход из функции и возвращение значения осуществляется инструкцией **return**:

```
return возвращаемое-значение;
```

При этом возвращаемое значение приводится к типу возвращаемого значения, указанному в заголовке функции.

Если приведение типа аргумента (к типу параметра) или типа возвращаемого значения (в контексте выражения, заключающего в себе вызов функции) невозможно, то компилятор сообщит об этом ошибкой компиляции.

Чистая функция [*pure function*] — функция, результат которой зависит только от значений аргументов, а вызов не влечет побочных эффектов.

Поведение чистых функций легче анализировать, оно более предсказуемо, а это снижает вероятность ошибок. По этой причине желательно при наличии возможности определять функции как чистые.

Рассмотрим некоторые простые примеры.

```
int sqr(int x) {
  return x*x;
}
```

Имея функцию sqr, такое выражение, как

$$((n+m)^2 + (n+k)^2)^2,$$

несложно записать в виде следующего кода:

```
sqr(sqr(n + m) + sqr(n + k))
```

В простейших случаях, вроде только что рассмотренного, можно представлять себе вызов функции как подстановку кода функции в место вызова. При этом параметры инициализируются значениями аргументов (*это условный код, не предусмотренный синтаксисом C++*):

```
sqr(n + m) →
({int x = n + m; return x*x;})
```

Пример 3.1. Пример функции с несколькими параметрами

```
// Линейная интерполяция.
float lerp(float x0, float x1, float a) {
  return (1.f - a)*x0 + a*x1;
}
```

Например, выражение

```
lerp(10, 20, 0.4f)
```

имеет значение 14.f — говорят «*вызов* lerp(10, 20, 0.4f) *вернет значение 14 типа float*».

Функции sqr и lerp являются чистыми функциями[57].

В объявлениях функции можно указывать произвольные имена параметров или даже не указывать их вовсе:

```
// Имя параметра можно не писать:
int sqr(int);

// Можно указать не все имена:
float lerp(float a, float b, float);
```

Полезно выбирать имена, подсказывающие читателю смысл соответствующих параметров.

Определение и объявление функции позволяют задать значения параметров по умолчанию, добавив инициализатор после (возможно, отсутствующего) имени параметра:

```
float lerp(float a, float b, float = 0.5f);
```

После такого объявления можно не передавать последний параметр в вызове lerp — в случае его отсутствия компилятор подставит 0.5f. Например, теперь lerp(1, 3) вернет 2.f.

Инициализаторы можно добавлять только с конца списка параметров. Следующие объявления функций будут отвергнуты компилятором:

```
// Ошибочные объявления значений по умолчанию:
int f(int a = 0, int b);
int g(int x, int y = 1, int);
```

Сигнатура функции [*function signature*] — кортеж типов параметров функции.

Например, наша функция sqr имеет сигнатуру (**int**), а lerp — (**float**, **float**, **float**). Имена параметров и их значения по умолчанию (если они есть) никак не влияют на сигнатуру.

---

[57] На самом деле, увы, не совсем так. Во-первых, любая функция, которая может повлечь *неопределенное поведение*, не может считаться чистой. Во-вторых, операции с плавающей точкой могут вызывать побочные эффекты (например, сохранение информации об ошибке в глобальной переменной errno).

В одной и той же *области видимости* мы можем объявить или определить несколько функций, имеющих одно имя, при условии, что эти функции имеют различные сигнатуры.

Перегрузка функции [*function overload*] — ситуация соответствия одному имени нескольких функций. Каждая из таких функций называется **перегруженной функцией**.

Термин «перегруженная функция» трудно назвать удачным, поскольку с формальной точки зрения все эти функции никак друг с другом не связаны, просто так получилось, что у них одинаковые имена. Но раз у них одинаковые имена, то невозможно только по имени определить конкретную функцию, а значит, «перегруженным» следовало бы называть имя функции.

Выбор конкретной функции из набора перегруженных осуществляется компилятором в точке вызова таким образом, чтобы набор типов аргументов наиболее точно соответствовал сигнатуре выбранной функции. Не всегда этот выбор однозначен, спорная ситуация приводит к ошибке компиляции.

Пример 3.2. Пример перегруженной функции

```
#include <cmath> // чтобы была доступна функция sqrt
// Вычислить длину вектора на плоскости R^2.
float len(float x, float y) {
  return std::sqrt(x*x + y*y);
}
// Вычислить длину вектора в пространстве R^3.
float len(float x, float y, float z) {
  return std::sqrt(x*x + y*y + z*z);
}
```

Если у второй функции в примере задать значение по умолчанию z = 0, то вызов len(x, y) станет неоднозначным и приведет к ошибке компиляции.

Композиция функций — передача в качестве аргумента одной функции результата вызова другой (или той же самой) функции.

Например, возведение в четвертую степень можно организовать композицией sqr с самой собой:

```
sqr(sqr(x)) // x^4
```

Пример 3.3. Более сложная композиция

```
// Билинейная интерполяция.
float lerp2(
    float x00, float x01,
    float x10, float x11,
    float ax = 0.5f, float ay = 0.5f) {
  return lerp(
    lerp(x00, x01, ax),
    lerp(x10, x11, ax), ay);
}
```

Функциональная декомпозиция — представление решения задачи в виде композиции функций.

Функция является основным строительным блоком программ. Функциональная декомпозиция является основным методом программирования.

В C++ имеется один особый тип — **void**[58]. Особость этого типа в том, что его множество значений пусто. А это значит, что не может существовать значений этого типа, нельзя определить переменную этого типа.

Естественно тогда спросить, зачем нужен такой тип.

В языке C данное ключевое слово играет роль формального обозначения отсутствия у функции параметров или возвращаемого значения:

```
// Процедура в C.
void greet(void) {
  printf("Greetings!");
}
```

[58] Англ. *void* — «пустой, недействительный».

В C++ писать **void** в списке параметров не требуется. Однако если мы хотим «функцию» (процедуру), не возвращающую значения, то в качестве возвращаемого типа надо указать **void**:

```
// Процедура в C++.
void greet() {
  cout << "Greetings!";
}
```

Выход из такой функции может быть осуществлен инструкцией **return** без указания какого-либо значения:

```
  return; // покинуть процедуру.
```

Впрочем, такой **return** может содержать в себе выражение типа **void**, например, вызов другой процедуры:

```
void world() { cout << "world!"; }
void hello_world() {
  cout << "hello, ";
  return world();
}
```

Возможно даже явное приведение к типу **void** выражения любого типа:

```
  (void)(cout << "this is nothing")
```

Эта конструкция говорит компилятору, что данное выражение надо вычислить (применить связанные с его вычислением побочные эффекты), а результат — отбросить. На практике такая конструкция встречается редко.

Задания для самопроверки

У Определите истинность следующих высказываний:

- Препроцессор — это автомат.
- Препроцессор выполняется после компоновщика.
- Определение можно считать частным случаем объявления.

- Корректная программа на C++ может содержать определения двух функций main.
- Функция является подвидом переменной.
- Прототип функции и объявление функции суть одно и то же.
- Любая функция обязана принимать и возвращать одно значение.
- Если функция ничего не принимает, то в ее определении круглые скобки после ее имени можно не ставить.
- Функция

```
float pi() {
  return 3.14159265f;
}
```

  является чистой.

- Функция

```
float cube(float x) {
  return x*x*x;
}
```

  является чистой.

- Функция

```
int inc(int a) {
  std::cout << "inc " << a << '\n';
  return a + 1;
}
```

  является чистой.

- Если чистая функция имеет возвращаемый тип **void**, то можно считать, что ее тело пусто (не содержит инструкций кроме пустых).

- Значения по умолчанию параметров функции требуется указывать, начиная с начала списка параметров.
- В C++ тип возвращаемого значения функции входит в ее сигнатуру.
- Пусть функция sqr возвращает квадрат своего аргумента. Тогда вызов sqr(sqr(sqr(x))) вернет x^6.
- Выбор конкретной перегруженной функции осуществляется компилятором в месте вызова путем сопоставления типов аргументов и сигнатур функций.
- Значения параметров по умолчанию можно указать только в объявлении функции.

У Пусть дана функция

```
float sqr(float x) {
  return x*x;
}
```

Запишите на C++ с ее помощью выражение (использовать sqr четыре раза)

$$((x+y)^4-(u+v)^2)^{-2}.$$

У Запишите формулу, вычисляемую функцией lerp2.

У С помощью функций lerp и lerp2 определите функцию lerp3 одиннадцати параметров, вычисляющую выражение

$$\begin{aligned}(1-a_z)((1-a_y)((1-a_x)x_{000}+a_x x_{001})+\\ +a_y((1-a_x)x_{010}+a_x x_{011}))+\\ +a_z((1-a_y)((1-a_x)x_{100}+a_x x_{101})+\\ +a_y((1-a_x)x_{110}+a_x x_{111})).\end{aligned}$$

Глава 4

Ветвления и циклы

4.1. Ветвления

Простейшим способом организации программы является **последовательность инструкций** — список действий, выполняемых по порядку:

```
инструкция-1
инструкция-2
инструкция-3
```

Инструкции выполняются в порядке их записи, все побочные эффекты полностью вступают в силу до перехода к следующей инструкции.

Определения в этом смысле также можно рассматривать как инструкции. Например, определение переменной можно рассматривать как инструкцию создания (и, возможно, инициализации) переменной. Все побочные эффекты инициализатора полностью вступают в силу до перехода к следующей инструкции или к следующему определению.

Процесс программирования нередко сводится к определению того, какие действия необходимо выполнить и в каком порядке, после чего их следует выписать в нужном порядке в виде программы («императивное программирование»).

Блок — то же, что и **составная инструкция** [*compound statement*], т. е. последовательность инструкций (возможно, пустая), рассматриваемая как одна инструкция.

В C++ блок оформляется взятием последовательности составляющих его инструкций в фигурные скобки { }:

```
{
  инструкция-1
  инструкция-2
  инструкция-3
}
```

Пустой блок { } является пустой инструкцией.

Переменные, определенные в блоке, видны только в этом блоке и существуют до закрывающей блок скобки }:

```
int xyz = 312;
{
  float xyz = 1.311f;
  cout << xyz << '\n'; // 1.311f
}
cout << xyz << '\n';    // 312
```

Итак, последовательность инструкций предполагает, что выполняться будут все указанные действия. Однако нередко требуется выполнить либо одно действие, либо другое действие, в зависимости от некоторого условия («альтернатива» или «ветвление» [*branching*]). Для этого предусмотрена инструкция **if**:

```
if (условие)
  инструкция-T
else
  инструкция-F
```

Здесь условие есть некоторое выражение. Если оно *истинно*, то будет выполнена инструкция-T (называемая **прямой ветвью** [*consequent branch*]). Если оно *ложно*, то будет выполнена инструкция-F (называемая **альтернативной ветвью** [*alternative branch*]).

Так или иначе, вначале **if** вычисляет условие и применяет все побочные эффекты данного вычисления, а уже затем выполняет либо ту, либо другую инструкцию.

Чтобы выполнить в качестве прямой или альтернативной ветви сразу несколько инструкций, следует их объединить в блок.

Допустима сокращенная форма без альтернативной ветви:

```
if (условие)
  инструкция-T
... // не else
```

Она эквивалентна подстановке пустой инструкции в качестве альтернативной ветви:

```
if (условие)
  инструкция-T
else
  ;
...
```

Пример использования альтернативы в коде:

```
int age = 0;
cout << "What is your age? ";
cin >> age;
if (age < 18)
  cout << "You're too young!\n";
else
  cout << "Access granted.\n";
```

В качестве альтернативной ветви может выступать другая инструкция **if**. Каждый **else** связывается с ближайшим сверху доступным (например, не упакованным в блок) **if**.

Это позволяет организовать «каскад условий» с перебором условий в порядке их перечисления до обнаружения первого истинного. Последняя альтернативная ветвь здесь будет альтернативой всем условиям и выполнится, если все условия оказались ложными. Ее наличие, конечно, не является обязательным:

```
if (условие-1)
  ветвь-1 // если условие-1
else if (условие-2)
  ветвь-2 // если не условие-1, но условие-2
else if (условие-3)
  ветвь-3 // если не условия 1 и 2, но условие-3
... // еще сколько-то else-if
else
  альтернативная-ветвь // если все условия оказались ложными
```

Модифицируем пример использования альтернативы в коде, добавив каскад условий:

```
int age = 0;
cout << "What is your age? ";
cin >> age;
if (age < 18)
  cout << "You're too young!\n";
else if (age > 80)
  cout << "You're too old!\n";
else
  cout << "Access granted.\n";
```

Чтобы разобраться с тем, как записываются и интерпретируются условия, необходимо ввести еще один тип.

Булевский[59] **тип** является типом значений логических выражений и имеет множество значений { *ложь*, *истина* } и набор операций { *и*, *или*, *не* }.

В C++ данный тип задается ключевым словом **bool**. Для его значений и операций также предусмотрены ключевые слова: *ложь* — **false**, *истина* — **true**, *и* — **and**, *или* — **or**, *не* — **not**.

Кроме того, булевские значения порождаются *операторами сравнения*. Сравнение на равенство записывается с помощью **двух** знаков «равно» ==, отличаясь таким образом от оператора присваивания =, который также может присутствовать в выражении, в том числе и условии. Важно их не путать, поскольку не всегда такие ошибки легко обнаружить.

[59] Англ. *Boolean* от фамилии английского математика Дж. Буля (1815–1864).

Оператор	Сравнение на
<	меньше
>	больше
<=	меньше или равно
>=	больше или равно
==	равенство
!=	неравенство

Операторы сравнения в C++

В C++ в случае приведения к числовому типу **false** приводится к значению 0, а **true** приводится к значению 1. Наоборот, приведение числа к булевскому типу (в том числе при использовании в качестве условия) эквивалентно сравнению на неравенство с нулем — нуль приводится к **false**, все прочие значения приводятся к **true**. Так что, например:

```
if (-100) // то же, что if (true)
  всегда-выполняется
if (0)    // то же, что if (false)
  никогда-не-выполняется
```

Необходимо осознавать, что условие является всего лишь выражением, значение которого определяет выбор того или иного действия. И как выражение, условие *вычисляется* и *имеет значение*, которое при желании можно сохранить или вывести на экран. Его также можно использовать в качестве подвыражения при вычислении какого-либо числового значения:

```
float a = 0, b = 1, x = 0;
cin >> a >> b >> x;
cout << ((a <= x) + (b <= x));
```

В C++ допустимо в инструкциях, принимающих условие, определять переменную в качестве условия: ее значение после инициализации будет приведено к булевскому типу и использовано для осуществления выбора. Определенная таким образом переменная будет доступна под соответствующей инструкцией.

```
int arg = -1;   // какая-то переменная
if (float arg = 3.1416f)
  cout << arg; // что выведет?
cout << arg;    // что выведет?
```

Для объединения условий C++ предлагает набор *логических операторов*.

Оператор	Лог. запись	Смысл
!, **not**	$\neg$	не (отрицание)
&&, **and**	$\wedge$	и (конъюнкция)
||, **or**	$\vee$	или (дизъюнкция)

Логические операторы в C++

Ключевые слова **not**, **and** и **or** являются альтернативой записи !, && и ||, соответственно. Несмотря на наличие данных ключевых слов, на практике они почти никогда не используются.

Например, высказывание «не p или (q и r)» может быть записано в традиционных обозначениях как $\neg p \vee (q \wedge r)$ или даже $\overline{p} \vee qr$, а на C++ множеством способов:

```
not p or q and r  // приоритет and выше, чем or
!p or q && r
not p || q and r
```

Все эти записи равнозначны, но на практике в подавляющем числе случаев мы встретим одну из следующих форм:

```
!p || q && r
!p || (q && r)
```

Итак, оператор «не» (отрицание) заменяет истину на ложь, а ложь — на истину.

Оператор «и» (конъюнкция) возвращает истину только в том случае, если истинны оба его операнда. В противном случае он возвращает ложь.

Можно заметить, что если первый операнд конъюнкции — ложь, то и результат будет ложь, независимо от значения второго операнда, поэтому его можно не вычислять. Именно так работает оператор && в C++: сначала вычисляется первый операнд. Если он ложен, то второй операнд не вычисляется, а результат оператора — ложь. Иначе вычисляется второй операнд, который и определяет конечный результат. Такой способ вычисления называется *вычисление по короткой схеме* [*short-circuit computation*].

Оператор «или» (дизъюнкция) возвращает ложь только в том случае, если ложны оба его операнда. В противном случае он возвращает истину.

Так же, как и конъюнкция, дизъюнкция в C++ вычисляется по короткой схеме: если первый операнд истинен, то второй операнд уже не вычисляется, результат — истина.

Это свойство (вычисление по короткой схеме) очень важно: && и || гарантированно вычисляют свои операнды по порядку и все побочные эффекты вычисления левого операнда вступают в силу до вычисления правого операнда, который вычисляется только в определенном случае, а значит, в ином случае может быть бессмысленным (*если делитель не нуль* и *частное больше x...*).

Конъюнкция часто возникает при проверке попадания числа в заданный диапазон. В C++ не поддерживаются «цепочечные» сравнения, поэтому то, что в математических обозначениях можно записать как $a \leqslant x < b$ или $x \in [a, b)$, в C++ требует связки «и»:

```
a <= x && x < b
```

Неприятным открытием может стать тот факт, что запись `a <= x < b` может быть воспринята компилятором не то что без ошибок, но даже без замечаний, однако смысл данного сравнения будет совсем не тот, что у $a \leqslant x < b$!

Начиная с C++17 инструкция **if** позволяет определить переменные отдельно от условия (или в дополнение к условию):

```
if (int x, y; cin >> x >> y)
  cout << "Pair (" << x << ", " << y << ")\n";
```

Таким способом можно определить набор переменных одного типа. Эти переменные видны только внутри инструкции **if**, включая зависимую ветвь **else**.

Инструкция **if** позволяет записать выбор того или иного действия, но не позволяет включить выбор внутрь выражения. В C++ имеется особый оператор, позволяющий включить выбор непосредственно в выражение. Этот оператор называют *тернарным оператором*, поскольку это единственный в C++ оператор, связывающий три операнда:

условие ? значение-если-истина : значение-если-ложь

Иногда тернарный оператор весьма удобен. Так, функцию, возвращающую модуль числа, можно записать следующим образом:

```
float abs(float x) { // модуль числа
  return x < 0? -x: x;
}
```

Тернарный оператор также позволяет организовать каскад перебора условий:

```
int sgn(int x) { // знак числа
  return x < 0? -1
       : 0 < x? +1
       : 0;
}
```

4.2. Циклы

Цикл [*loop*] — явная организация повторения набора инструкций (называемого **тело цикла** [*loop body*]). В языках программирования высокого уровня цикл обычно выражается с помощью специальной языковой конструкции.

Итерация[60] [*iteration*] — очередное исполнение тела цикла. Таким образом, выполнение цикла можно представлять себе как последовательность итераций.

Простейший вид цикла — **вечный цикл** [*infinite loop*].

```
for (;;)
  инструкция
```

Данный вид цикла не предусматривает условия прекращения повторения. Это не означает, что он действительно должен повторяться вечно. Во-первых, пользователь или система могут прекратить выполнение программы извне «насильно». Во-вторых, может произойти передача управления вовне цикла с помощью механизма исключений (см. раздел 18.5). В-третьих, программист может предусмотреть выход внутри цикла. От второго варианта завершения это отличается тем, что обязательно требует наличия в теле цикла кода, выполняющего выход из него. Рассмотрим этот вариант подробнее.

Выйти из цикла можно, покинув с помощью инструкции **return** функцию, содержащую этот цикл.

Пример 4.1. Цикл с выходом посередине

```
// Вычисление кубического корня методом Галлея.
float halley_cbrt(float a) {
  bool up = a < -1.f || (0.f < a && a < 1.f);
  float x = a; // выбор начального приближения
  for (;;) {
    float x3 = x*x*x;
    float y = x*((x3 + 2*a)/(2*x3 + a));
    if (x == y || up != (x < y))
      return x; // выйти из функции (и цикла)
    x = y;
  }
}
```

Данный способ хорошо сочетается с функциональной декомпозицией и потому предпочтителен. Но не во всех случаях

[60] От лат. *iteratio* — «повторение», *itero* — «повторяю», *iterum* — «опять, снова».

удобно выделять цикл в отдельную функцию, поэтому имеется альтернативный способ: инструкция **break** прекращает выполнение цикла, в котором она была выполнена:

```
int n = -1;
cout << "Enter an integer between 0 and 10\n";
for (;;) {
  cin >> n;
  if (0 <= n && n <= 10)
    break; // выход посередине цикла
  cout << "Invalid input. Repeat, please.\n";
}
cout << "Entered: " << n; // break ведет сюда
```

Подобные комбинации **for**, **if** и **break** или **return** позволяют выразить любую циклическую конструкцию. Но в целях повышения выразительности и читаемости кода, способствующих уменьшению числа ошибок и упрощению поддержки и повторного использования кода, следует при наличии возможности использовать более специализированные виды циклов.

Отдельный интерес представляет **цикл чтения**, являющийся идиоматической конструкцией C++ на основе **for**:

```
for (определение-переменных; чтение-переменных;)
  работа-с-переменными
```

При этом «чтение-переменных» должно быть выражением, результатом которого является либо булевское значение «успех чтения» непосредственно, либо объект потока ввода, из которого производилось чтение переменных. Объект потока ввода в качестве условия автоматически приводится к булевскому значению «отсутствие ошибок ввода», благодаря чему при входе в тело цикла на очередной итерации мы знаем, что переменные действительно были успешно прочитаны.

Пример 4.2. Сумма чисел из потока ввода

```
float sum = 0.f;
for (float num; cin >> num;)
  sum += num;
cout << sum; // вывести сумму
```

Пример 4.3. Длина ломаной на плоскости

```
float len = 0.f;
if (float x0, y0; cin >> x0 >> y0) {
  for (float x1, y1; cin >> x1 >> y1;) {
    len += hypot(x1 - x0, y1 - y0);
    x0 = x1;
    y0 = y1;
  }
}
cout << len; // вывести длину ломаной
```

(Функция hypot является стандартной и объявлена в заголовочном файле **cmath**.)

Пример 4.4. Удалить пробельные символы

```
for (char ch; cin >> ch;)
  cout << ch;
```

В случае чтения символа (тип **char**) из потока ввода с помощью оператора >> по умолчанию пропускаются все пробельные символы. Если требуется читать все символы, то можно использовать функцию потока ввода get, что можно воплотить в виде той же идиоматической конструкции цикла чтения, поскольку имеется вариант данной функции, возвращающий объект потока аналогично оператору >>:

```
// Увеличиваем код прочитанного символа на 1.
for (char ch; cin.get(ch);)
  cout.put(char(ch + 1));
```

Функция put записывает символ и является симметричным аналогом функции get. Приведение к **char** в данном случае не является строго обязательным, так как put принимает только символьный тип, к которому тип **int** (тип выражения ch + 1) может быть приведен неявно. Но явное приведение в подобных случаях полезно, так как мы таким образом показываем, что действительно хотим привести значение типа **int** к значению типа **char** и избавляемся от возможного предупреждения компилятора из-за угрозы потери части представления.

Рассмотренные выше примеры являются частными случаями общей конструкции **for**, называемой также *циклом общего вида*, характерной для языка программирования C и языков, основанных на его синтаксисе. Данная конструкция имеет следующий вид:

```
for (инициализация-цикла; условие; инкремент)
  тело-цикла
```

Итак, заголовок цикла содержит три секции:

- Инициализация-цикла может содержать определение (однотипных) переменных или выражение и выполняется однократно перед началом цикла.

- Условие может содержать определение переменной или выражение и вычисляется перед каждой итерацией.

  Если значение переменной или выражения приводится к истине, то цикл продолжается, иначе — заканчивается.

  Если секция условия пуста, то считается, что условие всегда истинно.

- Инкремент может содержать выражение, которое вычисляется в конце каждой итерации.

Иными словами, данную конструкцию можно представить в форме вечного цикла следующим образом:

```
{ // Блок, так как все переменные, определенные
  // в заголовке, видны только внутри цикла.
  инициализация-цикла;
  for (;;) {
    if (условие) {
      тело-цикла
      инкремент;
    }
    else
      break;
  }
}
```

Наиболее часто встречающийся частный случай данного цикла — **цикл со счетчиком**. Например, можно перечислить целые числа от 0 до 10 и вывести их квадраты:

```
// Простейший цикл со счетчиком.
for (int i = 0; i < 11; ++i)
  cout << i << " squared = " << i*i << "\n";
```

Вывести квадраты в обратном порядке:

```
for (int i = 10; i > -1; --i)
  cout << i << " squared = " << i*i << "\n";
```

Если требуется передвигать значение не на единицу, то разумно использовать совмещенное присваивание:

```
// Нечетные числа от 1 до 100.
for (int i = 1; i < 100; i += 2)
  cout << i << ", ";
```

Более сложные конструкции могут использовать более одной переменной:

```
// Перечислить степени двойки.
for (int i = 0, p = 1; i < 20; ++i, p *= 2)
  cout << i << "-th power of 2 = " << p << "\n";
```

Бинарный оператор , не следует путать с запятой в качестве пунктуации, например, в списке параметров функции. Выражение (выражение-1, выражение-2) означает «вычислить выражение-1 и применить все связанные с ним побочные эффекты, затем вычислить выражение-2 и применить все связанные с ним побочные эффекты, а значением всего выражения сделать значение выражения-2». Значение выражения-1 будет потеряно.

Таким образом, выражение ++i, p *= 2 означает, что сначала мы увеличим i на единицу, затем удвоим значение переменной p, а значением этого выражения будет новое значение p (но здесь это значение никак не используется, нам нужны только побочные эффекты — изменение значений переменных).

Типичной ошибкой является попытка организовать *двойной цикл* с помощью одного цикла со счетчиком, определив в нем две переменные и поставив условия через запятую:

```
// Плохая таблица умножения.
for (int i=2, j=2; i < 10, j < 10; ++i, ++j)
  cout << i << "*" << j << "␣=␣" << i*j << "\n";
```

Попробуйте определить, что в реальности будет делать данный цикл. (Можно заменить запятую в условии на связку «и» &&, здесь это ничего не изменит.)

Поскольку для каждого значения одной переменной требуется перебрать множество значений другой переменной, то следует записать два цикла **for**. На каждой итерации *внешнего цикла* [*outer loop*] полностью, от начала и до конца, выполняется *внутренний цикл* [*inner loop*]:

```
// Таблица умножения.
for (int i=2; i < 10; ++i)     // внешний цикл
  for (int j=2; j < 10; ++j) // внутренний цикл
    cout << i << "*" << j << "␣=␣" << i*j << "\n";
```

Такой цикл называют **двойным**. Аналогично можно записать тройной цикл и цикл еще большей степени вложенности. Все циклы в таких конструкциях, кроме внешнего, называют **вложенными** [*nested*]. Попробуйте определить, сколько точек выведет следующий код:

```
for (int i = 0; i < 2; ++i)
  for (int j = 0; j < 3; ++j)
    for (int k = 0; k < 4; ++k)
      for (int l = 0; l < 5; ++l)
        cout.put('.');
```

Цикл с предусловием — цикл, в котором перед выполнением каждой итерации проверяется некоторое условие. Если мы попали в тело цикла, то условие было истинно. Если цикл завершился, то условие было ложно. Если условие ложно с самого начала, то тело цикла не выполнится ни разу.

В C++ цикл с предусловием записывается с помощью инструкции **while**:

```
while (условие)
  тело-цикла
```

Данная конструкция эквивалентна следующему циклу на основе **for**:

```
for (; условие; )
  тело-цикла
```

Как и в случае **for**, условие может быть выражением или определением переменной.

Цикл с постусловием — цикл, в котором сначала выполняется тело, затем проверяется условие продолжения. Таким образом, тело цикла выполнится, по крайней мере, один раз, даже если условие окажется ложным.

В C++ цикл с постусловием записывается с помощью комбинации **do-while**:

```
do
  тело-цикла
while (условие);
```

Данная конструкция эквивалентна следующему циклу на основе **for**:

```
for (;;) {
  тело-цикла
  if (условие)
    ; // пустая инструкция
  else
    break;
}
```

Циклы с постусловием встречаются сравнительно редко. Если написанный код имеет вид

```
фрагмент-кода
while (условие)
  фрагмент-кода
```

где «фрагмент-кода» повторяется, то естественным будет сократить эту запись до цикла с постусловием:

```
do
  фрагмент-кода
while (условие);
```

Иногда при написании циклов может пригодиться еще одна инструкция, которую мы ранее не затрагивали, а именно, инструкция **continue**. Если **break** передает управление за цикл, то **continue** пропускает оставшийся код текущей итерации (тела цикла ниже **continue**) и переходит сразу к следующей итерации. Если цикл содержит условие, то оно проверяется, как и при обычном исполнении тела цикла.

На практике **continue** встречается довольно редко. Конструкцию с **continue** всегда можно переписать без **continue**. Обычно для этого достаточно обратить условие **if** и расставить скобки {}. Тем не менее, укажем два типичных варианта применения **continue**.

Вариант первый: вместо пустой инструкции (пустого тела цикла), чтобы показать читающему код человеку, что автор не ошибся, оставив тело цикла пустым, а сделал это сознательно:

```
// Пропустить строку в потоке ввода.
for (char ch; cin.get(ch) && ch != '\n';)
  continue;
```

Вариант второй: для улучшения читаемости кода при наличии внутри тела цикла условий, предусматривающих какие-то вспомогательные действия и тем «нарушающих» основную линию действий или попросту вызывающих нагромождение элементов.

Пример 4.5. Без continue

```
// Четыре угла.
for (int dx = -1; dx < 2; ++dx) {
  if (dx != 0) {
    for (int dy = -1; dy < 2; ++dy) {
      if (dy != 0)
```

```
            cout << "(" << dx << ", " << dy << ")\n";
        }
    }
}
```

Пример 4.6. C continue

```
// Четыре угла.
for (int dx = -1; dx < 2; ++dx) {
  if (dx == 0) continue;
  for (int dy = -1; dy < 2; ++dy) {
    if (dy == 0) continue;
    cout << "(" << dx << ", " << dy << ")\n";
  }
}
```

Рассмотрим еще одну инструкцию — инструкцию безусловного перехода по метке **goto**. Она имеет следующий вид:

```
goto метка;
```

Метка ставится перед инструкцией, на которую осуществляется переход:

```
метка: инструкция
```

Вечный цикл через **goto**:

```
forever:
  cout << "Never ends.\n";
  goto forever;
```

Переход можно осуществлять только в пределах тела функции. Запрещается «перескакивать» определения переменных.

С помощью комбинации **if** и **goto** можно выразить любой цикл, но такие конструкции обычно затрудняют людей, читающих код, поэтому **goto** использовать без серьезных на то оснований не рекомендуется.

4.3. Перебор вариантов

Помимо **if-else** C++ располагает еще одной конструкцией для организации ветвления. Это инструкция **switch**. Данная

инструкция позволяет для заданного значения (которое мы будем называть *селектор*) осуществить переход на первую строчку, помеченную значением, совпадающим со значением селектора. Эти значения, включая селектор, должны быть целыми числами.

```
switch (селектор) {
case значение-1: инструкции-1
case значение-2: инструкции-2
...
default: инструкции-прочие
}
```

Если селектор — выражение, то данный код можно считать почти эквивалентным следующему условному коду:

```
do {
  auto __s = селектор;
  if (__s == значение-1) goto __case_1;
  if (__s == значение-2) goto __case_1;
  ...
  goto __default;
__case_1: инструкции-1
__case_2: инструкции-2
...
__default: инструкции-прочие
} while (0);
```

Конструкция **do-while** в данном случае подразумевает однократное исполнение кода, так как условие всегда ложно. Ее смысл заключается в том, что поскольку формально это цикл, его можно покинуть с помощью инструкции **break**.

Как и в случае условия **if**, селектор в **switch** может быть определением переменной, значение которой затем используется для перехода на соответствующую метку **case**. Данная переменная существует только внутри блока **switch**.

Несмотря на то что **case** выглядят как метки (и исторически вся конструкция происходит от популярного в языках низкого уровня «вычисляемого перехода» [*computed goto*]), они не могут использоваться как цель перехода инструкции **goto**. Впрочем,

если в заголовке **switch** не содержится определений, то **goto** может «запрыгнуть» внутрь блока **switch**, для этого просто потребуется там поставить отдельную метку.

Покинуть **switch** можно с помощью **break** (только сам блок **switch**), **goto**, **return** и даже **continue**, если **switch** находится внутри цикла.

Идея «перехода на метку **case**» проявляется в общей семантике «выбора»:

```
int x = 2;
switch (x) {  // выведет: 23
case 1: cout << '1';
case 2: cout << '2';
case 3: cout << '3';
}
```

Исполнение продолжается до конца блока **switch**, если не встретится ни одной инструкции перехода, поэтому обычно каждый **case** «закрывают» инструкцией **break**:

```
int x = 2;
switch (x) {  // выведет: 2
case 1: cout << '1'; break;
case 2: cout << '2'; break;
case 3: cout << '3'; break;
}
```

Потеря **break** внутри **switch** является одним из традиционных источников ошибок при программировании на C и C++.

Так как **switch** действует как переход по метке, запрещается определять переменные в самом блоке под **switch**. Но их всегда можно взять в дополнительный блок:

```
switch (val) {
// Один и тот же код для случаев 1 и 2:
case 1: case 2: cout << "1 or 2"; break;
case 3: {
  unsigned x = val; // можно: переменная в блоке
  cout << x;
} break;
case 4:
```

```
  unsigned y = val; // ошибка компиляции
case 5:
  cout << y; // какое возможное значение y?
}
```

Как и в случае инструкции **if**, инструкция **switch** позволяет определить набор однотипных переменных в необязательной секции перед селектором:

```
switch (определение-переменных; селектор) { ... }
```

C++ предоставляет возможность определить тип как набор номинальных значений (**перечислимый тип** [*enumerated type*]) с помощью ключевого слова **enum**[61]:

```
enum Color { Green, Yellow, Red };
```

Данное определение вводит новый тип Color, у которого есть три возможных значения: Green, Yellow и Red.

Обратите внимание на ; после закрывающей скобки. Определения типов в C++ позволяют сразу использовать их в качестве типа в определении переменных, однако чаще всего переменные этого типа определяются в другом месте, поэтому мы просто закрываем определение точкой с запятой.

Определение переменной типа Color с инициализацией:

```
Color color = Yellow;
```

Перечислимый тип естественным образом сочетается с инструкцией **switch**:

```
void report(Color color) {
  switch (color) {
  case Green:  cout << "green";  break;
  case Yellow: cout << "yellow"; break;
  case Red:    cout << "red";    break;
  }
}
```

Если тип нужен для определения одной-единственной переменной, то ему можно не давать имя:

[61] От англ. *enumeration* — «перечисление».

```
enum { Green, Yellow, Red } color = Yellow;
```

Более того, мы можем использовать константы, определенные внутри **enum** просто как целочисленные константы времени компиляции (они получают значения 0, 1, 2, ...). Соответственно, сам **enum** можно использовать для определения набора целочисленных констант (опять же имя типу в этом случае давать необязательно):

```
enum { Green, Yellow, Red };
cout << Green << Yellow << Red; // 012
```

Можно явно указать базовый целочисленный тип перечисления (по умолчанию **int** или *общий тип* значений констант), а также желаемые значения констант. Если значение не указать, то оно будет равно предыдущей константе, увеличенной на единицу, или нулю, если оно первое в перечислении.

```
enum : uint32_t { // 32-битное целое без знака
  One = 1,
  Mask_M = (One << 24) - 1,
  Mask_E = ((One << 8) - 1) << 24,
  Mask_S = One << 31
};
```

(О типе uint32_t и подобных рассказано на с. 97.)

Во всех приведенных выше примерах имена констант попадают в область видимости, в которой находится **enum**. Кроме того, эти константы неявно приводятся к базовому целочисленному типу перечисления. Иногда хочется избежать такого поведения. Для этого следует добавить после ключевого слова **enum** ключевое слово **class** (или **struct** — эффект одинаков):

```
enum class Day_of_week {
  Sunday, Monday, Tuesday, Wednesday,
  Thursday, Friday, Saturday };
```

Данным определением мы запрещаем неявное приведение констант к целочисленным типам и заключаем их в пространстве имен самого перечислимого типа (поэтому такой тип обязан иметь имя — иначе невозможно будет обратиться к опреде-

ленным в нем константам). Для обращения к вложенным именам используется оператор ::. Например:

```
bool is_weekend(Day_of_week day) {
  return day == Day_of_week::Sunday
      || day == Day_of_week::Saturday;
}
```

Задания для самопроверки

У Сравните следующие две конструкции. Эквивалентны ли они? Если нет, то в чем заключаются их отличия?

```
// Конструкция 1.
if (условие-1)
  if (условие-2)
    действие

// Конструкция 2.
if (условие-1 && условие-2)
  действие
```

У Сравните следующие две конструкции. Эквивалентны ли они? Если нет, то в чем заключаются их отличия?

```
// Конструкция 1.
if (условие-1)
  действие
if (условие-2)
  действие // то же самое, что выше

// Конструкция 2.
if (условие-1 || условие-2)
  действие
```

У Как компилятор понимает следующий код?

```
// Одна из типичных ошибок новичков.
if (условие);
  действие
```

У С помощью тернарного оператора запишите определение функции min от трех целочисленных значений.

У Напишите вечный цикл на основе **while**.

У Сколько восклицательных знаков выведет следующий код?

```
do cout << '!'; while (0);
```

У Пусть дано следующее определение:

```
enum { A = -2, B, D, C = 99, E };
```

Чему теперь равно D? Чему равно E?

У Пусть есть определение

```
enum class Yes_no { No, Yes, Undefined };
```

Используя **switch-case**, определите функцию

```
Yes_no yes_no(char);
```

которая возвращает для 'y' и 'Y' значение Yes, для 'n' и 'N' — No, во всех прочих случаях — Undefined.

У Напишите вечный цикл на основе **do-while**.

У С помощью цикла со счетчиком запишите код, выводящий факториалы чисел от 0 до 10.

У Почему следующий код зависает?

```
int n = 10, p = 1;
while (n > 0);
  p *= (n -= 2);
```

У Почему следующий код не выводит последовательность чисел, делящихся на семь?

```
int i = 1;
for (; i < 1000; ++i)
  if (i % 7 == 0);
    cout << i << "␣";
```

Как его исправить?

Как его улучшить?

У Определите, что делает следующий код:

```
if (char prev; cin.get(prev)) {
  cout.put(prev);
  for (char ch; cin.get(ch);)
    if (ch != prev)
      cout.put(prev = ch);
}
```

У Реализуйте вывод таблицы умножения именно как таблицы с помощью двойного цикла **for**). Для выравнивания чисел по столбцам можно использовать символ табуляции \t.

У Напишите программу, запрашивающую числа a_0, s, n, вычисляющую сумму арифметической прогрессии из n элементов с начальным элементом a_0 и шагом s непосредственно через сумму и с помощью формулы, затем выводящую разность этих двух значений. Проверьте, получается ли 0, если a_0 или s не являются целыми.

Замечание. Используйте для представления членов прогрессии числа с плавающей точкой (тип **float** или **double**).

У Реализуйте игру «Угадай число», в которой человек загадывает число от 1 до 10, а компьютер его угадывает за четыре шага.

Глава 5

Логические вычисления

5.1. Логика и множества

Скобка Айверсона — обозначение численного значения логического утверждения (0 для лжи, 1 для истины) через заключение его в квадратные скобки. Например:

$$\mathrm{sgn}(x) \equiv [x > 0] - [x < 0].$$

В C++ обычно допустимо использовать значение логического выражения в вычислениях непосредственно:

```
int sgn(int x) {
  return (x > 0) - (x < 0);
}
```

Индикатор множества S или **характеристическая функция множества** S — это функция вида $\mathbf{1}_S\colon X \mapsto B$, такая, что $\mathbf{1}_S(x) \triangleq [x \in S]$. Здесь X — некое надмножество множества S (т. е. $S \subseteq X$), а $B = \{0, 1\}$ (множество значений булевского типа).

Предикат — функция, областью значений которой является множество значений булевского типа. Можно сказать, что предикат отвечает на вопрос «да/нет».

Итак, с одной стороны, для любого множества A можно определить эквивалентный ему предикат — индикатор этого множества $\mathbf{1}_A(x) \triangleq [x \in A]$.

С другой стороны, любому предикату $p(x)$ соответствует некоторое множество $P = \{\, x \in X \mid p(x) \,\}$, на котором предикат истинен. Любой предикат является индикатором своего множества истинности: $p(x) \equiv \mathbf{1}_P(x)$.

Таким образом, предикаты и множества можно считать эквивалентными понятиями.

Особняком стоит вопрос о свойствах множества всех предикатов на заданном множестве.

Пусть множество X конечно и $|X| = n$. Что можно сказать о множестве всех предикатов, определимых на множестве X? Итак, для любого элемента $x \in X$ некий предикат $p(x)$ может быть равен либо 0, либо 1. Так как множество X конечно, мы можем подменить его натуральным рядом $1, 2, \ldots, n$ и подменить предикаты композицией предиката и нумерации $\mathrm{p}(i) \triangleq p(x_i)$, $i = \overline{1,n}$. Теперь каждому предикату p взаимнооднозначно соответствует бит-строка $\mathrm{p}(1), \mathrm{p}(2), \ldots, \mathrm{p}(n)$ длины n. Поскольку каждый бит в бит-строке может быть либо 0, либо 1, то всего разных бит-строк длины n имеется 2^n.

Отсюда вывод: множество предикатов, определенных на конечном множестве размера n, тоже является конечным и имеет размер 2^n.

Так как множество предикатов изоморфно множеству подмножеств, то, значит, множество всех подмножеств некоторого конечного множества размера n также конечно и имеет размер 2^n. Помимо прочих, оно в качестве своих элементов включает пустое множество (всюду ложный предикат) и само исходное множество (всюду истинный предикат).

Булеан [*powerset*] 2^X множества X — множество всех подмножеств множества X.

Теперь рассмотрим случай бесконечного множества X.

Должно быть очевидно, что булеан бесконечного множества не может быть конечным множеством. Интуитивно понятно,

что булеан несчетного множества не может быть множеством счетным. Остается вопрос — является ли булеан счетного множества тоже счетным?

Для доказательства этого достаточно построить нумерацию натуральными числами булеана множества натуральных чисел $2^{\mathbb{N}}$. Для опровержения этого достаточно показать невозможность такой нумерации.

Булеан множества натуральных чисел изоморфен множеству всех предикатов, заданных на натуральных числах.

Предположим, что существует нумерация множества предикатов натуральных чисел натуральными числами. Это значит, что для всякого $n \in \mathbb{N}$ мы можем выписать первые n предикатов (в порядке их нумерации). Выпишем их значения для чисел $1, 2, \ldots, n$. Получим таблицу $n \times n$ из нулей и единиц.

Например, пусть $n = 7$ и таблица имеет вид:

№ предиката	1	2	3	4	5	6	7
1	**0**	1	0	1	0	0	0
2	0	**1**	0	0	0	1	0
3	1	0	**1**	1	0	1	1
4	0	0	1	**1**	1	0	0
5	0	0	0	0	**0**	0	1
6	1	1	1	0	0	**1**	0
7	1	1	0	1	1	1	**0**
?	1	0	0	0	1	0	1

Эта таблица может быть заполнена 0 и 1 произвольным образом. Главная диагональ выделена полужирным шрифтом. Взяв в каждой колонке элемент главной диагонали и обратив его, получим новый предикат, последовательность значений которого выписана в строчке, помеченной вопросительным знаком.

Этот новый предикат не совпадает ни с одним из пронумерованных предикатов, поскольку для любого $i \in \mathbb{N}$ предикат i не совпадает с построенным предикатом в значении для

числа i. Данная конструкция известна как *диагональный аргумент Кантора*[62].

Таким образом, какую бы нумерацию предикатов мы не выбрали, всегда можно построить как минимум еще один предикат, который не имеет номера в этой нумерации, что означает, что пронумеровать все предикаты невозможно, и значит, множество всех предикатов натуральных чисел (или любого счетного множества) несчетно.

Следствие 1. Множество действительных чисел $\mathbb{R}$ несчетно. Следует из того, что отрезок $[0, 1]$ изоморфен множеству предикатов натуральных чисел.

Следствие 2. В данной формальной конструкции не все предикаты натуральных чисел (и действительные числа) *определимы*, поскольку определение — строка, а множество всех строк счетно.

Теоретико-множественные операции — операции, операндами которых являются множества. Примеры: объединение, пересечение, дополнение множеств.

Благодаря эквивалентности предикатов и множеств можно указать соответствие между теоретико-множественными операциями и логическими операциями над индикаторами множеств.

Пусть X — *универсальное множество*, *универсум* (множество, содержащее все рассматриваемые элементы и являющееся надмножеством всех рассматриваемых множеств).

Пусть A, B и C — произвольные подмножества X. Дополнение множества A до X обозначим через $\overline{A} \triangleq X \setminus A$. Запись вида $\mathbf{1}_A \wedge \mathbf{1}_B$ следует понимать как определение функции $(\mathbf{1}_A \wedge \mathbf{1}_B)(x) \triangleq \mathbf{1}_A(x) \wedge \mathbf{1}_B(x)$, где $x \in X$ — произвольный аргумент.

Симметрическая разность множеств может быть определена через объединение и пересечение: $A \triangle B \triangleq (A \cup B) \setminus (A \cap B) \equiv$ $\equiv (A \cap \overline{B}) \cup (\overline{A} \cap B)$.

---

[62] Г. Кантор (1845–1918) — немецкий математик. Ввел данную конструкцию, доказывая существование несчетных множеств.

	Логика	Множества	
ложь	0	$\varnothing$	пустое множество
истина	1	X	универсум
предикат	$\mathbf{1}_A$	A	множество
отрицание	$\neg\mathbf{1}_A$	$\overline{A}$	дополнение
	$\equiv 1 - \mathbf{1}_A$	$\equiv X \setminus A$	
конъюнкция	$\mathbf{1}_A \wedge \mathbf{1}_B$	$A \cap B$	пересечение
дизъюнкция	$\mathbf{1}_A \vee \mathbf{1}_B$	$A \cup B$	объединение
разность	$\mathbf{1}_A - \mathbf{1}_B$	$A \setminus B$	разность
	$\equiv \mathbf{1}_A \wedge \neg\mathbf{1}_B$	$\equiv A \cap \overline{B}$	
исключ. *или*	$\mathbf{1}_A \veebar \mathbf{1}_B$	$A \triangle B$	симм. разность
импликация	$\mathbf{1}_A \to \mathbf{1}_B$	$\overline{A} \cup B$	
	$\equiv \neg\mathbf{1}_A \vee \mathbf{1}_B$		
эквиваленция	$\mathbf{1}_A \leftrightarrow \mathbf{1}_B$	$\overline{A \triangle B}$	
	$\equiv \neg(\mathbf{1}_A \veebar \mathbf{1}_B)$		

Соответствие между логическими и
теоретико-множественными операциями

В то же время логические высказывания, использующие предикаты, можно отобразить на логические высказывания о соответствующих множествах.

Знаки $\to$, $\leftrightarrow$ и $\Rightarrow$, $\Leftrightarrow$ использованы разные, чтобы различать соответствующие булевские операции и логические связки. Эта разница становится ясна как раз при переводе высказываний на язык множеств.

Иногда может быть удобным рассматривать значение индикатора как число (собственно 0 или 1). Тогда связка $\Leftrightarrow$ может трактоваться как утверждение о равенстве (операция эквиваленция — как операция сравнения на равенство), а имплика-

Логика	Множества
$\mathbf{1}_A(x)$	$x \in A$
$\neg\mathbf{1}_A(x)$	$x \notin A$
$(\forall x)\mathbf{1}_A(x)$	$A = X$
$(\exists x)\mathbf{1}_A(x)$	$A \neq \varnothing$
$\neg(\exists x)\mathbf{1}_A(x)$	$A = \varnothing$
$(\forall x)\mathbf{1}_A(x) \Rightarrow \mathbf{1}_B(x)$	$A \subseteq B$
$(\forall x)\mathbf{1}_A(x) \Leftrightarrow \mathbf{1}_B(x)$	$A = B$

Соответствие между некоторыми высказываниями

ция — как сравнение $\leqslant$, символ которого даже визуально напоминает теоретико-множественный аналог[63] $\subseteq$.

Также легко убедиться, что:

$$\mathbf{1}_A(x) \wedge \mathbf{1}_B(x) \equiv \mathbf{1}_A(x) \cdot \mathbf{1}_B(x) \equiv \min\{\mathbf{1}_A(x), \mathbf{1}_B(x)\},$$
$$\mathbf{1}_A(x) \vee \mathbf{1}_B(x) \equiv \max\{\mathbf{1}_A(x), \mathbf{1}_B(x)\},$$
$$\mathbf{1}_A(x) \veebar \mathbf{1}_B(x) \equiv \mathbf{1}_A(x) \oplus \mathbf{1}_B(x) \equiv [\mathbf{1}_A(x) \neq \mathbf{1}_B(x)],$$

где $\oplus$ означает операцию *сложение по модулю 2* (данное название даже нередко используется как синоним названия *исключающее или* [*exclusive or, xor*]).

Ниже приведена таблица логических высказываний, справедливых в классической математической логике, и их аналогов из теории множеств.

Примеры

Пусть есть круг радиуса 3 с центром в точке $(1, 1)$ и квадрат со стороной 6 с центром в точке $(-1, -1)$. Индикаторы этих

[63] К сожалению, множества, в отличие от чисел, не являются *линейно упорядоченными*.

Логика	Множества
$\neg 1 = 0$	$\overline{X} = \varnothing$
$1 \vee a = 1$	$X \cup A = X$
$1 \wedge a = a$	$X \cap A = A$
$0 \vee a = a$	$\varnothing \cup A = A$
$0 \wedge a = 0$	$\varnothing \cap A = \varnothing$
$a \vee \neg a = 1$	$A \cup \overline{A} = X$
$a \wedge \neg a = 0$	$A \cap \overline{A} = \varnothing$
$\neg\neg a = a$	$\overline{\overline{A}} = A$
$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$	$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$	$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
$\neg(a \vee b) = \neg a \wedge \neg b$	$\overline{A \cup B} = \overline{A} \cap \overline{B}$
$\neg(a \wedge b) = \neg a \vee \neg b$	$\overline{A \cap B} = \overline{A} \cup \overline{B}$
$a \to b = \neg b \to \neg a$	$A \subseteq B = \overline{B} \subseteq \overline{A}$
$(a \to b \wedge b \to c) \Rightarrow (a \to c)$	$(A \subseteq B \wedge B \subseteq C) \Rightarrow (A \subseteq C)$
$(a \leftrightarrow b \wedge b \leftrightarrow c) \Rightarrow (a \leftrightarrow c)$	$(A = B \wedge B = C) \Rightarrow (A = C)$

Истинные высказывания

двух фигур можно записать как:

$$\mathbf{1}_{\text{кр}}(x, y) \triangleq [(x-1)^2 + (y-1)^2 \leqslant 9],$$
$$\mathbf{1}_{\text{кв}}(x, y) \triangleq [\max\{\, |x+1|, |y+1| \,\} \leqslant 3].$$

На рис. 5.1–5.3 приведены изображения фигур, индикаторы которых получены различными комбинациями $\mathbf{1}_{\text{кр}}$ и $\mathbf{1}_{\text{кв}}$.

5.2. Целочисленные типы C++

Помимо типа **int**, язык C++ предлагает еще ряд встроенных целочисленных типов. Все они разделены на две основные категории: *числа без знака* (*беззнаковые*) [*unsigned numbers*] и *числа со знаком* [*signed numbers*]. Диапазон значений чисел без

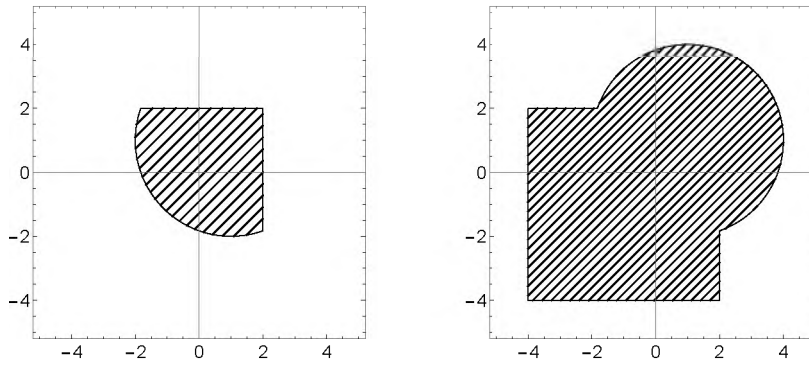


Рис. 5.1. $1_{\mathbf{кр}} \wedge 1_{\mathbf{кв}}$ и $1_{\mathbf{кр}} \vee 1_{\mathbf{кв}}$

знака не включает отрицательные элементы, начинаясь с нуля. Диапазон значений чисел со знаком содержит как неотрицательные, так и отрицательные числа.

Мы можем явно указать, что хотим тип чисел со знаком, с помощью ключевого слова **signed**:

```
signed int n = -100;
```

На деле это ключевое слово почти не используется, поскольку «число со знаком» есть вариант по умолчанию для всех целочисленных типов кроме **char**, т. е. **signed int** есть просто синоним для **int**.

Тип **char** занимает особое место. Хотя это и целочисленный тип, но стандарт не относит его ни к числам со знаком, ни к числам без знака. Тип **signed char** не является синонимом **char** даже в том случае, когда множества значений этих типов совпадают.

Ключевое слово **unsigned** позволяет указать тип чисел без знака:

```
// Значение переменной не может быть отрицательным:
unsigned int n = 100;
```

Таким образом, типы **unsigned int** и **int** — разные, и множества их значений различны.

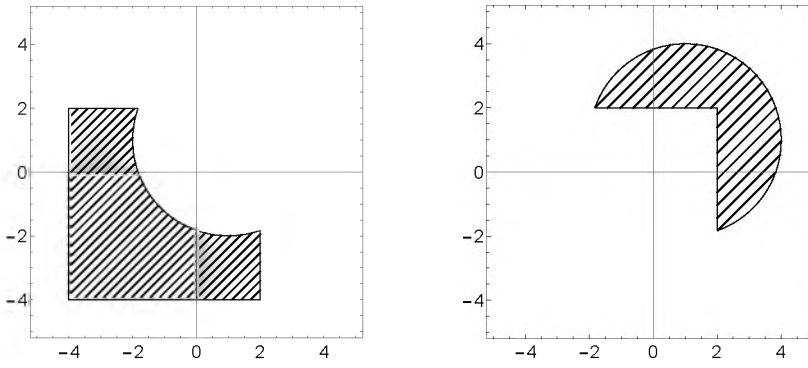


Рис. 5.2. $\neg 1_{\mathbf{кр}} \wedge 1_{\mathbf{кв}}$ и $1_{\mathbf{кр}} \wedge \neg 1_{\mathbf{кв}}$

Ключевое слово **int** можно не указывать:

```
unsigned u = 10; // то же, что unsigned int u = 10;
signed s = -10;  // то же, что int s = -10;
```

Тип **unsigned char** не является синонимом **char** даже в том случае, когда множества значений этих типов совпадают. Множество значений **char** совпадает либо со множеством значений **signed char**, либо со множеством значений **unsigned char**.

Кроме «знаковости» целочисленные типы различаются по ширине представления в двоичных разрядах. Все встроенные типы имеют фиксированную ширину представления и заранее определенные диапазоны представимых значений. Но конкретные значения этих параметров определяются системой и компилятором C++, стандарт лишь накладывает некоторые ограничения на их возможный выбор.

Итак, всего есть пять градаций ширины, большинство из которых могут совпадать, хотя обычно поддерживается четыре разных варианта:

char $\subseteq$ **short** $\subseteq$ **int** $\subseteq$ **long** $\subseteq$ **long long**.

Формально ключевые слова **short** и **long** являются модификаторами и предполагают применение к **int**, но так как **int** можно опустить, его часто не пишут. Поэтому, например, **long**,

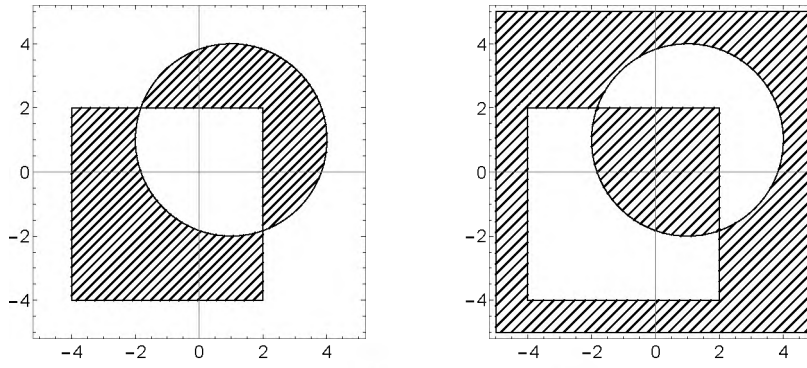


Рис. 5.3. $1_{\mathbf{кр}} \veebar 1_{\mathbf{кв}}$ и $\neg(1_{\mathbf{кр}} \veebar 1_{\mathbf{кв}})$

signed long, **long int** и **signed long int** суть разные названия одного и того же типа чисел со знаком.

Разрядность **int** без модификаторов часто совпадает с разрядностью **short** или с разрядностью **long**. Предполагается, что **int** соответствует наиболее «естественному» для данной платформы целочисленному типу. Арифметика со значениями более узких типов (**char** и **short**) производится после приведения значений к типу **int**.

Тип	Бит	min	max
signed char	8	$-2^7 = -128$	$2^7 - 1 = 127$
unsigned char	8	0	$2^8 - 1 = 255$
signed short	16	$-2^{15} = -32\,768$	$2^{15} - 1 = 32\,767$
unsigned short	16	0	$2^{16} - 1 = 65\,535$
signed long	32	$-2^{31} \approx -2.1 \cdot 10^9$	$2^{31} - 1$
unsigned long	32	0	$2^{32} - 1 \approx 4.3 \cdot 10^9$
signed long long	64	$-2^{63} \approx -9.2 \cdot 10^{18}$	$2^{63} - 1$
unsigned long long	64	0	$2^{64} - 1 \approx 1.8 \cdot 10^{19}$

Размеры и диапазоны значений на платформе Win32

Размер представления любого типа (кроме **void**), включая целочисленные, можно получить с помощью оператора

sizeof. (Попытка вычислить **sizeof void** есть ошибка компиляции.) Размер измеряется в «байтах», поэтому **sizeof char** всегда равен 1. Само это значение принадлежит некоторому встроенному типу чисел без знака, доступному по имени size_t, объявленному в заголовочном файле **cstddef**.

Оператор **sizeof** можно применять к именам типов:

```
// Сколько char поместится в int:
cout << sizeof(int);
```

или к переменным и вообще выражениям. В последнем случае этот оператор возвращает размер типа переменной или выражения (само выражение не вычисляется):

```
char ch = 'a';
// Размер int?
cout << sizeof(ch + ch);
```

Значение **sizeof**(a) есть константа времени компиляции.

У Что выведет следующий код? Почему?

```
cout << sizeof((void)(cout << "b"), 'a');
```

Для оформления констант типа **unsigned** и целочисленных типов более широких, чем **int**, предусмотрены суффиксы (см. таблицу). Например, 1 имеет тип **int**, а 1u имеет тип **unsigned**.

Тип	Суффиксы
signed int	нет
unsigned int	u или U
signed long	l или L
unsigned long	ul или UL
signed long long	ll или LL
unsigned long long	ull или ULL

Суффиксы записи целочисленных констант

Если в выражении есть операнды разных целочисленных типов не у́же **int**, то результат вычисляется как значение наиболее широкого типа. Если в выражении есть операнды разной

«знаковости» [*signedness*] одной ширины, то результат вычисляется как число без знака. Иногда это приводит к неожиданностям (например, значение -1 приводится к наибольшему допустимому значению беззнакового типа):

```
for (unsigned n = 10; n > -1; --n)
  cout << n << '\n'; // ни разу не выполнится
for (unsigned n = 10; n <= -1; --n)
  cout << n << '\n'; // вечный цикл!
```

Итак, следует избегать сравнений чисел без знака с числами со знаком. Вообще, ситуация со смешиванием чисел без знака и чисел со знаком является традиционным источником порой труднообнаружимых ошибок в программах на C и C++. Современные компиляторы обычно способны выдавать предупреждения в таким случаях.

Иногда предлагают не использовать числа без знака[64], кроме некоторых особых случаев, однако стандартная библиотека использует числа без знака для представления размеров структур данных. Недостатком чисел со знаком является обилие ситуаций неопределенного поведения (по стандарту), что, вероятно, связано с желанием поддерживать разные возможные реализации представления отрицательных чисел.

Переполнение (**наверх** [*overflow*], **вниз** [*underflow*]) — ситуация получения результата операции, не принадлежащего множеству значений типа. «Наверх» — получено значение, большее максимального представимого. «Вниз» — получено значение, меньшее минимального представимого.

Важно знать, что поведение при переполнении чисел со знаком объявлено стандартом C++ как неопределенное — компилятору дозволяется полагать, что такая ситуация никогда не происходит.

В то же время поведение при возникновении переполнения в операциях с числами без знака определено и реализуется как

[64] Авторы языка Java, например, вовсе отказались от встроенных типов чисел без знака.

оборачивание [*wrapping around*], т. е. операция выполняется в арифметике по модулю 2^b, где b есть разрядность соответствующего типа числа без знака. Поэтому, в частности, вычитание 1 из нуля без знака дает наибольшее представимое целое данного типа.

Значение одного целочисленного типа может быть преобразовано в значение любого другого целочисленного типа. При этом в общем случае не гарантируется сохранность этого значения: при обратном преобразовании можем получить значение, отличное от исходного. Это свойство очевидным образом следует из-за различия представимых разными типами множеств целых чисел.

Правила приведения целочисленных типов следующие:

- Если значение исходного типа принадлежит множеству значений целевого типа, то оно сохраняется.

- Если при приведении к типу чисел со знаком исходное значение не принадлежит множеству значений целевого типа, то результат определяется реализацией (стандарт его не регламентирует).

- Если при приведении к типу чисел без знака исходное значение не принадлежит множеству значений целевого типа, то результат определяется значением по модулю 2^b, где b — разрядность целевого типа. Это очень важное правило, благодаря которому отрицательные числа при приведении к беззнаковому типу приводятся к двоичному представлению в дополнительном коде (см. далее).

- Если приводится тип ме́ньшей разрядности к типу бо́льшей разрядности, то в случае, если число неотрицательное, недостающие старшие разряды заполняются нулями. Отрицательные числа получают соответствующее представление в расширенном типе со знаком, которое затем (если требуется) уже приводится к беззнаковому типу той же ширины.

Стандартный заголовочный файл **climits** определяет ряд констант, позволяющих узнать характеристики целочисленных типов данной реализации:

- CHAR_BIT — количество бит в **char** (не менее 8);
- *_MIN — минимум целочисленного типа;
- *_MAX — максимум целочисленного типа.

Вместо * следует подставить конкретное слово из списка:

- CHAR — тип **char**;
- SCHAR — тип **signed char**;
- UCHAR — тип **unsigned char**;
- SHRT — тип **signed short**;
- USHRT — тип **unsigned short**;
- INT — тип **signed int**;
- UINT — тип **unsigned int**;
- LONG — тип **signed long**;
- ULONG — тип **unsigned long**;
- LLONG — тип **signed long long**;
- ULLONG — тип **unsigned long long**.

Стандартный заголовочный файл **cstdint** предоставляет синонимы целочисленных типов заданной разрядности (вместо X ниже следует подставить 8, 16, 32 или 64):

- int$X$_t (опциональные) — целое со знаком разрядности X бит;

• uint$X$_t (опциональные) — целое без знака разрядности X бит;

• int_fast$X$_t — целое со знаком разрядности не меньше X бит;

• uint_fast$X$_t — целое без знака разрядности не меньше X бит;

• int_least$X$_t — целое со знаком наименьшей доступной разрядности не меньше X бит;

• uint_least$X$_t — целое без знака наименьшей доступной разрядности не меньше X бит;

• intmax_t — целое со знаком наибольшей доступной разрядности;

• uintmax_t — целое без знака наибольшей доступной разрядности;

• intptr_t (опциональный) — целое со знаком, способное вместить адрес переменной;

• uintptr_t (опциональный) — целое без знака, способное вместить адрес переменной.

Типы, помеченные как «опциональные», могут не предоставляться конкретной реализацией. Если uint$X$_t и int$X$_t предоставляются, то они имеют разрядность ровно X бит.

Если int$X$_t предоставляются, то они используют представление отрицательных чисел в дополнительном коде (см. ниже).

Типы int_fast$X$_t и uint_fast$X$_t предлагаются как типы, достаточные для представления целых чисел разрядности X, операции с которыми производятся наиболее быстро на данной платформе (оптимизация по скорости).

Типы int_least$X$_t и uint_least$X$_t предлагаются как типы, достаточные для представления целых чисел разрядности X,

имеющие наименьший размер (оптимизация по памяти, требуемой для размещения переменных).

5.3. Представление отрицательных чисел

C++ предполагает представление числа в виде последовательности двоичных разрядов (бит). Для неотрицательных целых чисел придумать способ кодирования не составляет труда — надо просто выписать представление в двоичной системе счисления. Например, один байт (8 двоичных разрядов) способен хранить двоичные коды от 00000000 (0) до 11111111 (255 как число без знака). Беззнаковые типы могут использоваться как строки бит фиксированной длины.

Однако не столь очевидно, как следует поступить с отрицательными целыми числами. Для кодирования целых чисел с диапазоном, включающим отрицательные числа, были придуманы различные способы. Стандарт C++ не регламентирует, какой способ должна использовать реализация (как правило, используется то представление, которое целевой процессор поддерживает аппаратно). Ниже даны описания четырех широко известных способов представления отрицательных чисел.

Прямой код [*signed magnitude representation*] предполагает выделение отдельного разряда под знак числа. Обычно это старший разряд. Если он равен нулю, то число (составленное из прочих разрядов) считается положительным, а если он равен единице, то отрицательным.

Обратный код [*one's complement*] представляет отрицательные числа в виде инверсии (каждый нуль заменяется единицей, а каждая единица — нулем) положительных аналогов.

Сдвинутый (или **смещенный**) **код** [*excess representation*] предполагает выбор некоторой константы b (*сдвиг* [*excess*]), на которую сдвигается (путем вычитания) представление числа, взятое как запись целого без знака. Обычно b выбирается равным середине диапазона беззнаковых (127 для 8-битного кода).

Дополнительный код [*two's complement*] получается из обратного кода добавлением единицы к инверсии, что позволяет складывать и вычитать числа как их беззнаковые представления.

В настоящее время для представления чисел со знаком почти повсеместно используется дополнительный код как наиболее удобный для аппаратной реализации (сложение, вычитание, сравнение на равенство не требуют проверки знака и могут выполняться так же, как для чисел без знака).

Биты	Прямой	Обратный	Дополн.	Сдвиг 127
0000 0000	0	0	0	−127
0000 0001	1	1	1	−126
0000 0010	2	2	2	−125
0111 1110	126	126	126	−1
0111 1111	127	127	127	0
1000 0000	−0	−127	−128	1
1000 0001	−1	−126	−127	2
1000 0010	−2	−125	−126	3
1111 1110	−126	−1	−2	127
1111 1111	−127	−0	−1	128

Интерпретация 8-битного представления в разных кодах

Положительная часть множеств значений и соответствующих представлений в прямом, обратном и дополнительном кодах совпадает (просто двоичная запись числа). Проверка на отрицательность может быть сведена к проверке значения старшего бита (но в прямом и обратном коде помимо «обычных» отрицательных чисел еще может быть «отрицательный нуль»).

Общим преимуществом прямого и обратного кодов является симметричность диапазона значений, благодаря чему смена знака всегда определена. Однако это преимущество оборачивается недостатком в виде наличия двух нулей — «обычного»

и «отрицательного» (при этом должно быть истинно $0 = -0$, поскольку это два представления одного значения).

В дополнительном и сдвинутом кодах нет избыточности, и множества значений содержат на один элемент больше. Но и симметричности отрицательной и положительной частей также нет, что приводит к тому, что смена знака определена не для всех элементов множества значений. Например, в примере с 8-битными числами в дополнительном коде имеем $-(-128) = = 0-(-128) = 1-(-127) = 1+127 \to 0000\,0001_2+0111\,1111_2 = = 1000\,0000_2 \to -128$.

В прямом и дополнительном коде проверка на четность очень проста и не зависит от знака числа — достаточно проверить значение младшего бита. Можно обобщить это и для проверки на делимость прочими степенями двойки.

При увеличении (уменьшении) на 1 представления числа, интерпретируемого как запись числа без знака, значение увеличивается (уменьшается) на 1 во всех кодах, кроме прямого (кроме случая пересечения «границы знаков»). В прямом коде направление меняется при пересечении границы знаков.

В дополнительном и сдвинутом кодах мы имеем своего рода «закольцованность» множеств представимых значений, что обычно проявляется при переполнении.

Так как стандарт C++ не предполагает конкретного представления чисел со знаком, то опора в программе на то или иное представление остается на совести программиста. Конечно, мы можем знать, что архитектура процессора, для которой компилируется наша программа, определяет этот способ и все эффекты, связанные с переполнением, но требуется уделять внимание поведению компилятора в пограничных случаях.

Однако приведение отрицательных чисел к беззнаковым типам предполагает их представление в дополнительном коде, поэтому, например, -1 можно привести к беззнаковому типу, чтобы получить двоичное представление, в котором все биты равны единице (это будет максимальное представимое число).

5.4. Поразрядные операции

Поразрядными булевскими (или **побитовыми** [*bitwise*]) **операциями** называют операции, предполагающие применение булевских операций к каждой паре двоичных разрядов, стоящих в одной позиции, независимо от других разрядов.

C++ предлагает четыре оператора для выполнения поразрядных операций: &, |, ^ и ~ (см. таблицу).

Оператор	Операция	Слева	Справа	Результат
&	*и*	1100	1010	1000
|	*или*	1100	1010	1110
^	*искл. или*	1100	1010	0110
~	*не*	—	1010	0101

Поразрядные операции C++

Отметим, что поразрядное *исключающее или* обладает следующими свойствами (a и b — числа без знака; булевское *исключающее или* обладает аналогичными свойствами на булевских значениях):

- коммутативность: a ^ b ≡ b ^ a;
- ассоциативность: (a ^ b) ^ c ≡ a ^ (b ^ c);
- нильпотентность: a ^ a ≡ 0;

и может выполнять роль «управляемого отрицания»:

- 0 ^ a ≡ a, (−1) ^ a ≡ ~a.

У Чем отличается (−1) ^ a от (~0) ^ a?

Благодаря своим свойствам и простоте аппаратной реализации операция поразрядное *исключающее или* популярна в криптографии, а также методах проверки корректности и восстановления данных.

Из приведенных выше свойств легко выводится утверждение (a ^ b) ^ b ≡ a. Данное утверждение обосновывает применимость известного «трюка», позволяющего обменять значения двух целочисленных переменных без использования явной временной переменной.

```
// Пара переменных.
int a = 23, b = 42;

// Обычный обмен через промежуточную переменную:
int t = a;
a = b; // теперь a == 42
b = t; // теперь b == 23

// Трюк через исключающее или ("xor"):
a ^= b; // промежуточное значение a == 61
b ^= a; // теперь b == 42
a ^= b; // теперь a == 23

// Трюк через xor в одну строчку:
a ^= b ^= a ^= b;
```

На практике не следует применять данный прием без действительной необходимости: на современных процессорах такой код работает медленнее, чем использование промежуточной переменной (которую компилятор часто может удалить на этапе оптимизации).

Кроме булевских поразрядных операторов, имеются операторы сдвига бит, выполняющие «сдвиг» значений бит влево или вправо по разрядной сетке. Разряды, «выдвигаемые» за пределы фиксированного набора разрядов представления числа, отбрасываются (**линейный сдвиг**, также называемый *логическим*). Освобождающиеся справа разряды заполняются нулями. Линейный сдвиг целого числа a влево на n бит записывается как a << n.

В случае линейного сдвига вправо возможно заполнение освобождающихся левых разрядов строго нулями либо значением самого старшего разряда (такой сдвиг называется **ариф-**

метическим). Линейный (возможно, арифметический) сдвиг целого числа a вправо на n бит записывается a >> n.

Для чисел без знака операторы сдвига выполняют линейный сдвиг с заполнением нулями. В случае чисел со знаком сдвиг вправо нередко реализуют как арифметический сдвиг, но стандарт не требует этого.

Сдвиг на n бит не определен, если n превосходит разрядность числа. Сдвиг влево на n бит числа со знаком a считается неопределенным, если значение $a \cdot 2^n$ не принадлежит множеству значений типа, либо если $a < 0$.

Сдвиг вправо неотрицательного числа a на n бит равен $\lfloor a \cdot 2^{-n} \rfloor$[65]. В случае отрицательного числа результат зависит от конкретной реализации. При использовании арифметического сдвига и дополнительного кода (наиболее распространенный случай) также имеем $\lfloor a \cdot 2^{-n} \rfloor$ (например, $(-6) >> 2$ дает -2, т. е. $\lfloor \frac{-6}{4} \rfloor$).

Помимо линейного сдвига иногда используется сдвиг **циклический** [*rotation*], предполагающий заполнение освобождающихся с одного конца разрядов значениями разрядов, выдвигаемых с другого конца. C++ не поддерживает циклический сдвиг непосредственно, но его несложно записать через сдвиг линейный (оптимизирующий компилятор способен заменить вызов такой функции на одну команду процессора):

```
// Циклический сдвиг вправо.
//    0 <= bits <= 32.
uint32_t rotate_right(uint32_t arg, unsigned bits) {
  return (arg << (32 - bits)) | (arg >> bits);
}
```

У Напишите аналогичную функцию rotate_left, выполняющую циклический сдвиг влево.

Ниже показаны результаты сдвига 8-битного слова на 0–4 бит вправо для циклического и линейного сдвигов.

---

[65] Округление к ближайшему целому, не превосходящему аргумент округления (*округление вниз*).

На	Циклич.	Линейн.	Арифм.
0	11001101	11001101	11001101
1	11100110	01100110	11100110
2	01110011	00110011	11110011
3	10111001	00011001	11111001
4	11011100	00001100	11111100

Сдвиг вправо

У Продолжите таблицу «Сдвиг вправо» для сдвигов на 5–8 бит.

Маской [*mask*] называют последовательность бит, указывающую, какие разряды требуется извлечь путем применения поразрядного u к другой последовательности бит. Соответствующие биты в маске установлены в 1, прочие — в 0.

Маску можно представлять себе в виде картонного шаблона, в котором прорезаны отверстия в определенных местах. При наложении такого шаблона на текст (вычислении поразрядного u) мы видим только те цифры, которые попали под отверстия — биты в тех позициях, где в маске стоят единицы.

Например, с помощью операции сдвига можно получить маску для одного бита с конкретным номером:

```
uint32_t(1) << n // 1 в позиции n, тип uint32_t
```

Применяя такую маску, очень легко вывести двоичное представление числа:

```
uint32_t x = 0xABCD9876;
for (unsigned n = 32; n != 0;) {
  uint32_t mask = uint32_t(1) << --n;
  cout.put(x & mask? '1': '0');
}
```

Следующий пример демонстрирует сложение пары чисел разрядности 128 бит, заданных парами 64-битных чисел (все без знака, арифметика по модулю 2^{128}).

```
uint64_t a_hi, a_lo; // первое число
uint64_t b_hi, b_lo; // второе число
uint64_t c_hi, c_lo; // результат
... // задать значения a и b
// Сложить верхние половинки.
c_hi = a_hi + b_hi;
// Сложить нижние половинки.
c_lo = a_lo + b_lo;
// Добавить перенос в случае переполнения.
c_hi += (c_lo < a_lo) | (c_lo < b_lo);
```

Ряд семейств процессоров (например, x86 и ARM) предлагают команду «сложение с переносом (от предыдущей операции)», что позволяет организовать эффективное сложение «длинных чисел» с автоматическим добавлением переноса — три последние строчки можно было бы выполнить всего лишь двумя командами процессора (не считая возможных пересылок значений между памятью и регистрами процессора).

Языки высокого уровня, включая C++, как правило, не располагают встроенными средствами для представления подобных конструкций, но есть небольшой шанс, что оптимизирующий компилятор «догадается» их применить.

Используя запись числа в системе счисления с основанием r, очень легко получить частное и остаток от деления этого числа на произвольную степень r^n. Для этого достаточно «разрезать» запись числа справа от разряда n (считая разряды с нуля). Так как компьютер представляет числа в двоичной системе счисления, то можно использовать поразрядные операции для вычисления частного и остатка от деления[66] на 2^n.

[66] Практический смысл в таких манипуляциях заключается в исключительной простоте поразрядных операций с точки зрения аппаратной реализации, поэтому иные процессоры могут не иметь команд для деления или даже для умножения, но иметь достаточный набор команд для поразрядных операций. Даже на современных «больших» процессорах умножение, не говоря о делении, вычисляются медленнее поразрядных операций.

Например, в случае $n = 5$ для некоторого 16-битного числа имеем:

число	0100 0111 0011 1010
маска частного	1111 1111 1110 0000
маска остатка	0000 0000 0001 1111
частное	0100 0111 001
остаток	0000 0000 0001 1010

Так как частное можно получить сдвигом, маска частного нам не понадобится. Маска остатка численно равна $2^n - 1$, поэтому ее можно получить с помощью сдвига.

```
// Исходное число.
uint32_t num;
// Порядок степени делителя.
unsigned n;
... // получить num и n
// Частное:
uint32_t quot = num >> n;
// Остаток:
uint32_t rem_mask = (uint32_t(1) << n) - 1
uint32_t rem = num & rem_mask;
```

Конечно, если есть частное, можно получить остаток и альтернативным способом:

```
uint32_t rem = num - (quot << n);
```

Но часто требуется только остаток, а маску можно вычислить заранее и использовать просто как константу.

Поскольку запись степени двойки в двоичной системе представляет собой единицу, за которой идет последовательность нулей, проверить, является ли положительное целое степенью двойки, очень просто:

```
// Является ли число степенью двойки.
bool is_pow2(unsigned n) {
  return ((n - 1) & n) == 0;
}
```

Данная функция определит нуль тоже как степень двойки, но на практике этим обычно можно пренебречь.

Вообще, операция

```
(n - 1) & n
```

обнуляет самый младший из единичных (*установленных* [*set*]) бит. Это позволяет, например, записать функцию, вычисляющую количество установленных бит (также называемое *вес Хэмминга* [*Hamming weight*]) следующим образом[67]:

```
int bitcount_Kernighan(unsigned n) {
  int count = 0;
  for (; n != 0; n &= n - 1)
    ++count;
  return count;
}
```

Есть еще более изощренные способы быстрого подсчета числа установленных бит, но мы здесь не будем их рассматривать. Следует отметить, что современные процессорные архитектуры, как правило, предлагают команду, выполняющую данное действие быстрее любого кода на основе операторов C++. Некоторые компиляторы предлагают для доступа к таким командам нестандартные средства. Например, GCC предлагает для этого функцию __builtin_popcount.

Последний пример, который мы здесь рассмотрим, — вычисление ближайшего сверху кратного заданной степени двойки. Данная задача нередко встречается в случаях, когда приходится вычислять размеры или сдвиги, которые должны быть кратны заранее известной степени двойки.

---

[67] Данный метод стал известен как «алгоритм Кернигана» благодаря книге Б. Кернигана и Д. Ритчи (см.: *Kernighan B. W., Ritchie D.* The C Programming Language. 2nd ed. Englewood Cliffs, NJ. 1988). По свидетельству Д. Кнута, этот метод был уже опубликован в 1960 г. П. Вегнером (см.: *Wegner P.* A technique for counting ones in a binary computer // Comm. ACM. 1960. № 3(5). P. 322).

Пусть заданы натуральные числа $P = 2^p$ и X. Необходимо вычислить наименьшее число $N \geqslant X$ такое, что $N = nP$ для некоторого $n \in \mathbb{N}$.

В случае, если остаток от деления X на P не равен нулю, то добавление к X максимально возможного такого остатка $(P - 1)$ дает перенос бита в разряд p. Если же остаток равен нулю, то переноса не будет. Поэтому, если обнулить младшие p бит суммы $(X + (P - 1))$, то получим искомое:

```
// Вместо unsigned можно подставить любой
// беззнаковый тип,
// число pow2 должно быть степенью двойки.
unsigned nceil2(unsigned x, unsigned pow2) {
  unsigned pow2m1 = pow2 - 1;
  return (x + pow2m1) & ~pow2m1;
}
```

Задания для самопроверки

У Докажите, что множество целых чисел $\mathbb{Z}$ является счетным.

У Докажите, что множество пар целых чисел $\mathbb{Z}^2$ является счетным.

У Какой вид имеет индикатор произвольной полуплоскости на $\mathbb{R}^2$?

У Как проверить четность числа, используя оператор &?

У Как извлечь значение бита с заданным номером?

У Как обнулить бит с заданным номером?

У Как обратить значение бита с заданным номером?

У С помощью поразрядных операций и без использования циклов и ветвлений реализуйте функцию, выполняющую «смешивание» двух целых a и b по заданной маске s. Результатом этой операции должно быть значение r, каждый бит r_i которого равен либо соответствующему биту a_i, если установлен соответствующий бит маски s_i, либо соответствующему биту b_i в противном случае.

У Напишите функцию, обменивающую местами 16-битные половинки 32-битного числа.

У Напишите функцию, обменивающую местами байты 32-битного числа.

У Как двумя операторами записать действие «установить самый младший из нулевых бит»?

У Пусть есть **int** a, для каких значений a результат a % 2 отличается от результата a & 1?

У Как вычислить *расстояние Хэмминга* между двоичными представлениями двух целых чисел? Расстоянием Хэмминга называется количество позиций, в которых соответствующие элементы представлений различны.

У Изучите функцию parity и объясните, что она вычисляет и как работает:

```
bool parity(uint32_t bits) {
  bits ^= bits >> 1;  // соседние биты
  bits ^= bits >> 2;  // соседние пары бит
  bits ^= bits >> 4;  // соседние четверки бит
  bits ^= bits >> 8;  // соседние байты
  bits ^= bits >> 16; // соседние пары байт
  return (bits & 1) != 0;
}
```

У Изучите функцию round_up_to_pow2 и объясните, что она вычисляет и как работает:

```
uint32_t round_up_to_pow2(uint32_t x) {
  --x;
  x |= x >> 1;  // Группа наложений-сдвигов эффективно
  x |= x >> 2;  // заменяет последовательность вида:
  x |= x >> 4;  // x |= x » 1; x |= x » 2; ...
  x |= x >> 8;  // x |= x » 31;
  x |= x >> 16; // На каждое удвоение разрядности
  return x + 1; // достаточно добавлять одну строчку
}
```

У Отдельные биты могут использоваться как признаки наличия элементов из некоторого конечного множества в подмножестве (представленном последовательностью бит). Так, если есть множество $X = \{0, 1, \ldots, 7\}$ (универсум), то 8-битный байт может служить представлением произвольного подмножества $M \subseteq X$. Если $i \in M$, то i-й бит байта установлен в единицу (в противном случае — в ноль).

Указанный способ представления подмножеств задает их естественную нумерацию и позволяет выполнять теоретико-множественные операции над конечными множествами из универсумов до 64 элементов (uint64_t) с помощью поразрядных операций.

Заполните в следующей таблице колонку «C++».

Множество или действие	C++
пустое множество ($\varnothing$)	
универсум (X)	
проверка принадлежности ($i \in A$)	
проверка на подмножество ($A \subseteq B$)	
— строгое подмножество ($A \subset B$)	
дополнение ($X \setminus A$)	
пересечение ($A \cap B$)	
объединение ($A \cup B$)	
разность ($A \setminus B$)	
симметрическая разность ($A \triangle B$)	

Глава 6

Процедурное программирование

6.1. Константы и ссылки

Начнем с ключевого слова, которое является одним из наиболее часто используемых ключевых слов в C++. Это ключевое слово — **const** («константа»).

Его следует использовать в тех случаях, когда значение некоторой переменной не предполагается изменять. Да, формально это все равно «переменная», хотя и считается, что после инициализации ее значение неизменно. Компилятор постарается проверить, пытаемся ли мы изменить значение этой переменной, и выдаст ошибку компиляции, если обнаружит такую возможность. Эта ошибка компиляции будет означать ошибку в наших рассуждениях или коде. Кроме того, явное объявление неизменяемых значений константами часто помогает компилятору производить более эффективный машинный код.

Ключевое слово **const** является модификатором типа и применяется к типу, стоящему слева от него. Если слева ничего нет, то компилятор ищет тип справа от **const**, и многие программисты обычно ставят это ключевое слово первым, если это возможно. В данной книге мы будем придерживаться более

регулярного правила: **const** всегда ставится справа от модифицируемого типа.

Следующий код можно использовать, чтобы посмотреть, как выглядит ошибка компиляции при попытке заменить значение константы значением, введенным пользователем:

```
float const pi = 3.141593f;
std::cout << pi << '\n'; // можно
std::cin >> pi; // ошибка компиляции!
```

Иногда бывает так, что мы хотим временно «защитить» некоторую переменную от возможного изменения. Для этого можно воспользоваться стандартной функцией std::as_const из заголовка **utility**.

Данная функция возвращает *ссылку*.

Ссылка [*reference*] — значение, являющееся псевдонимом другого значения. Одно значение может иметь множество ссылок, указывающих на него. Имя переменной можно считать ссылкой на представление значения этой переменной.

В C++ выполняются следующие правила:

- ссылка не может непосредственно указывать на другую ссылку;
- ссылка обязательно должна указывать на какое-то значение (обязательно должна быть инициализирована);
- сама ссылка не может быть изменена (переназначена на другое значение), т. е. является константой;
- тип ссылки (*ссылочный тип*) на значение типа T записывается как T&. Множеством его значений является множество переменных типа T.

Любые действия со ссылкой трактуются компилятором как действия, которые будут выполняться над объектом, к которому эта ссылка привязана.

```
int n = 100;
int & r = n;        // r — ссылка на n
```

```
r *= 2;
std::cout << n; // 200
int & r2 = r;    // r2 — ссылка на n!
r2 *= 2;
std::cout << n; // 400
// Ошибка компиляции:
int & & r3 = r; // ссылка на ссылку?
```

Ссылка на изменяемое значение может быть неявно приведена к ссылке на константу:

```
int n = 100;
int const & r = n;
std::cout << r; // 100
n *= 2;
std::cout << r; // 200
// Ошибка компиляции:
r *= 2; // нельзя изменить константу
// Ошибка компиляции:
int & const x = n; // ссылка и так константа
```

Казалось бы, зачем нам второе имя переменной? Причин может быть, по крайней мере, три:

1. Переменная имеет слишком длинное, неудобное название. Привязав к ней ссылку, мы получим более удобное, короткое локальное название. При этом мы можем не указывать тип переменной, вместо него можно использовать ключевое слово **auto**:

   ```
   auto & short_name =
     some_namespace::some_long_long_name;
   ```

2. Выбор объекта привязки ссылки может происходить во время исполнения программы и зависеть от некоего условия. Пример:

   ```
   int a = 0, b = 0;
   cin >> a >> b;
   // Привязать к b, если a < b, иначе — к a:
   int & max = a < b? b: a;
   ```

```
max = 42;
cout << "a = " << a << "; b = " << b << '\n';
```

3. И, наконец, главное. Ссылочный тип можно использовать для объявления параметров функции:

```
// Передача параметра по значению.
void by_value(int n) {
  // n — локальная переменная функции by_value
  n = 42; // изменение внутренней переменной
}

// Передача параметра по ссылке.
void by_reference(int & n) {
  // n — псевдоним внешней переменной
  n = 23; // изменение внешней переменной
}

// Тест.
int main() {
  int x = 0;
  by_value(x);
  cout << x; // 0
  by_reference(x);
  cout << x; // 23
}
```

Передача в функцию переменной по ссылке позволяет изнутри функции изменить эту переменную и является одним из обычных способов организации возвращения из функции более одного значения в C++. В этом случае есть определенная асимметрия: одно значение особое — собственно значение, возвращаемое из функции через **return**. Прочие значения могут быть возвращены через присваивание внешним переменным, переданным функции по ссылкам.

Ссылку можно также возвращать из функции:

```
// Возвращает одну из переданных переменных.
int & max(int & a, int & b) {
```

```
  return a < b? b: a;
}

int main() {
  int x = 7, y = 9;
  max(x, y) /= 2;
  cout << x << ' ' << y;
}
```

Не следует возвращать ссылку на переменные, не существующие после выхода из функции.

Ключевое слово **auto** по умолчанию «захватывает» значение, снимая ссылки:

```
int a = 10, b = 20;
auto & x = max(a, b); // int&
x = 30; // b = 30
auto y = max(a, b);    // int
y = 40; // b все еще 30
// y — отдельное целое.
```

6.2. Неопределенное поведение

В языке C++ важную роль играют понятия **неопределенное поведение** [*undefined behavior, UB*] и **определяемое реализацией поведение** [*implementation-defined behavior, IB*], характеризующие действия, результаты которых не определяются стандартом языка.

Когда некоторое действие объявляется как порождающее UB, это означает, что программист не должен полагаться на какой-то определенный результат — все зависит от выбора компилятора в данном конкретном случае и особенностей платформы, причем разработчики платформы и компилятора не обязаны указывать в документации последствия такого действия. Компилятор может считать, что ситуации UB просто невозможны. Например, компилятор GCC известен тем, что при определенных условиях просто выбрасывает из программы участки, зависящие от неопределенного поведения.

В случае IB разработчик компилятора должен выбрать некоторую, разумную с его точки зрения, реализацию и описать это в документации.

К сожалению, программа, опирающаяся на конкретное поведение на данной платформе с данным компилятором, строго говоря, не является переносимой. Часто использование конструкций, эффект которых заявлен как IB или UB, является неосознанным из-за невнимательности, недостатка опыта или знаний программиста. Если, например, поведение программы различается в отладочной (debug, оптимизация машинного кода компилятором выключена) и окончательной (release, оптимизация включена) сборках, то, скорее всего, виноват код, порождающий UB.

Ярким примером UB и ошибочного кода является повторное использование (в том числе повторное изменение) изменяемой переменной при вычислении выражения, когда относительный порядок вычисления подвыражений не определен:

```
++a -= b;                          // UB
cout << a << a++ << ++a;  // UB
```

Порядок вычисления аргументов функции в точке вызова тоже не определен:

```
int sum(int a, int b) {
  return a + b;
}
...
  int a = 1;
  cout << sum(++a, ++a);  // UB
```

Вызывающим UB также объявлено любое арифметическое действие с целыми числами со знаком, результат которого не может быть представлен в используемом машинном представлении числа (кроме приведения беззнаковых или чисел большей ширины).

Следующая функция возвращает **false**, если прибавление единицы не приводит к переполнению. В то же время переполнение **int** объявлено UB:

```
bool is_max_int(int x) {
  return x > x + 1;
}
```

Таким образом, оптимизирующий компилятор вообще может не вызывать эту функцию, а подставить в место вызова **false**!

```
int x = 1;
// Потенциально вечный цикл!
while (!is_max_int(x))
  ++x;
```

Для исключения возможных ситуаций UB следует выполнять поразрядные операции и сдвиги только со значениями беззнаковых типов — там они полностью определены.

Некоторые другие ситуации неопределенного поведения:

- целочисленное деление (включая взятие остатка) на нуль;
- умножение или деление в случае непредставимости результата (в том числе, частного при взятии остатка), например, ситуация, когда один из операндов — наименьшее представимое отрицательное целое, а другой — -1, по-разному обрабатывается на разных платформах;
- бесконечный цикл без возможности побочных эффектов:

  ```
  while (1) {} // зависает?
  cout << "done";
  ```

- возврат из функции ссылки на значение, которое прекращает существование после завершения функции (например, на локальную переменную);
- выход из функции без возвращения значения (кроме main и возвращаемого типа **void**);
- использование значения неинициализированной переменной (часто случающаяся ошибка).

Во многих сомнительных случаях (не только собственно UB) современные компиляторы способны генерировать предупреждения о возможной ошибке, поэтому рекомендуется включить вывод предупреждений и внимательно читать сообщения компилятора.

6.3. Процедурное программирование

Императивное программирование [*imperative programming*] — определение программы в виде явной[68] последовательности действий. Действия выражаются в форме инструкций, влекущих побочные эффекты, которые, собственно, и составляют видимую извне работу программы.

Концепция императивного программирования исторически тесно связана с программированием в машинных кодах реальных компьютеров и в простейшей версии предполагает наличие всего лишь двух базовых конструкций:

- нумерованного списка инструкций (выполняемых по порядку);
- условного перехода вида:

  if условие **goto** номер-инструкции

  Безусловный переход легко выразить через условный: достаточно поставить истину в качестве условия.

Этих конструкций, в принципе, достаточно для выражения любого алгоритма[69], но они неудобны при написании больших программ.

---

[68] От лат. *imperativus* — «повелительный», *impero* — «велю, повеливаю».

[69] Для получения всех арифметических действий в этом случае достаточно уметь выполнять, например, увеличение значения на единицу и обнуление значения. В качестве условия — проверку на равенство двух значений.

Принципиальным нововведением можно считать понятие *подпрограммы* [*subroutine*] или *процедуры*.

Процедура[70] [*procedure*] — именованный участок программы, который можно «вызвать» из другого места программы. Под вызовом понимается переход на начало процедуры с (автоматическим) возвращением обратно в место вызова после исполнения процедуры.

Рекурсия[71] [*recursion*] — вызов функцией самой себя (напрямую или опосредованно).

Рекурсивное определение — определение, ссылающееся на себя само.

Так как в реализациях ранних языков программирования возврат нередко осуществлялся прописыванием адреса[72] возврата непосредственно в код вызываемой процедуры, то процедура не могла вызывать себя (напрямую или опосредованно) — рекурсия не была возможна. Если рекурсия невозможна или не применяется, то аргументы процедуре можно передавать через выделенные для этого глобальные переменные, т. е. формально вызов не обязан «уметь» передавать и возвращать данные. Хотя на практике, конечно, удобнее определять параметры в заголовке процедуры, а не где-то в ином месте программы, избегая таким образом возможных конфликтов имен. Эта возможность была уже в самых ранних языках программирования высокого уровня в 1950-е годы.

В C++ вместо термина *процедура* используется (в широком смысле как синоним) термин *функция*. В узком смысле *процедура* — это функция, не возвращающая значений (в качестве типа возвращаемого значения указан **void**).

---

[70] От лат. *procedo* — «продвигаюсь».

[71] От лат. *recursio* — «возвращение», *recurro* — «возвращаюсь», *re-* — приставка со значением обращения или повторения действия, *curro* — «бегу, двигаюсь».

[72] Расположение соответствующей команды процессора в памяти программ. Можно считать его неким глобальным номером инструкции.

Процедурное программирование [*procedural programming*] — императивное программирование, дополненное конструкцией «процедура» и принципом «сверху-вниз».
Разработка «сверху-вниз» [*top-down design*] заключается в последовательном разбиении задач на подзадачи.

Итак, мы имеем некую задачу, которую должна решать наша программа. Процедурное программирование предполагает, что каждой (под)задаче должна соответствовать своя процедура. Собственно метод процедурного программирования может быть выражен в виде следующего списка действий:

- Мы определяем главную процедуру (в C++ это функция main — стандартная точка входа) как пустую.

- Разбиваем задачу на подзадачи.

- Определяем для каждой подзадачи свою процедуру, выполненную как *заглушку* [*stub*]. Под заглушкой понимается процедура, тело которой еще не решает задачу, а выводит (на экран или в файл) сообщение, характеризующее вызов этой процедуры (как минимум, название процедуры), либо вообще ничего не делает. Если она должна возвращать значение, то она возвращает какое-нибудь допустимое значение.

- Пишем код главной процедуры, обращаясь к процедурам, решающим подзадачи. Данный пункт обычно выполняется одновременно с предыдущим, потому что бывает сложно сразу точно определить удобный набор параметров каждой процедуры, отвечающей своей подзадаче.

- Отлаживаем главную процедуру, анализируя протокол, формируемый вызовами процедур-заглушек.

- Повторяем данную операцию для каждой подзадачи и соответствующей ей процедуры, пока не будет достигнут уровень элементарных действий (процедур, состоящих из

одной инструкции). Каждая процедура по возможности отлаживается независимо.

- В конце (или ближе к концу) разработки программа отлаживается целиком.

Структурное[73] **программирование** [*structured programming*] — вариант процедурного программирования, предполагающий запрет на явный переход по метке.

Структурное программирование использует такие понятия, как «блок» (составная инструкция с ограничением области видимости, вложенность блоков демонстрируется в тексте программы отступами), «ветвление» (**if-else**), «цикл со счетчиком», «цикл с предусловием», «цикл с постусловием». Инструкции не имеют меток. Наряду с этим обычно вводится понятие гетерогенного типа и далее «алгебраического типа данных» (будет рассмотрено позднее).

Любой алгоритм, который можно выразить в рамках классического императивного программирования, можно выразить, используя три конструкции: последовательность, ветвление и цикл с предусловием (данный результат известен как «теорема структурного программирования»[74]).

Языки C и Pascal изначально базировались на концепции структурного программирования, хотя **goto** была сохранена. Fortran «воспринял» структурное программирование в версии Fortran 90. Ряд более новых языков, например, Java и Python, не имеют инструкции безусловного перехода по метке.

---

[73] Данный перевод является исторически традиционным, хотя правильнее было бы перевести англ. *structured* как «упорядоченное», на контрасте с понятиями *unstructured programming* — «неупорядоченное программирование» и *literate programming* — «грамотное программирование».

[74] См.: *Bohm C., Jacopini G.* Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules // Comm. ACM. 1966. № 9(5). P. 366–371.

6.4. Модульное программирование

Модульное программирование [*modular programming*] — концепция, предполагающая явное разделение программы на **модули** [*modules*]. Каждый модуль решает строго очерченный круг задач и состоит из двух частей: *интерфейса* [*interface*] и *реализации* [*implementation*].

Модули могут импортировать другие модули. При этом импортирующий модуль имеет доступ только к интерфейсу импортируемого модуля, но не к реализации. Данный принцип называется *сокрытием деталей реализации* и облегчает возможности по изменению реализации каждого модуля независимо от прочих при условии сохранения интерфейсных частей.

Цель модульного программирования — *разделение ответственности* [*separation of concerns*] частей программы. Общие элементы выделяются в отдельные модули, что позволяет снизить (или элиминировать) дублирование функциональности[75] в разных частях программы и сделать ее разработку более управляемой, а структуру — более ясной. Каждым модулем может заниматься отдельная команда разработчиков.

Модули представляют собой крупномасштабные элементы программы, поэтому модульное программирование может свободно сочетаться с другими концепциями: структурным, функциональным и объектно-ориентированным программированием, и не зависит от них.

Рассмотрим возможности языка C с точки зрения поддержки модульного программирования.

Модуль реализуется как один или несколько заголовочных файлов (интерфейс) и одна единица трансляции (реализация). Модуль, импортирующий другой модуль, подключает соответствующий заголовок (интерфейс) с помощью директивы include.

---

[75] Необходимо, впрочем, отметить, что чрезмерное увлечение ликвидацией дублирования кода может приводить к значительному усложнению структуры программы и замедлению ее разработки.

Пример 6.1. Интерфейс модуля 1

```
// module_1.h
#ifndef MODULE_1_H_INCLUDED
#define MODULE_1_H_INCLUDED
// Необходимые директивы include.
...
// Определение типов и констант.
...
// Объявления переменных и функций.
...
#endif//MODULE_1_H_INCLUDED
```

Такая конструкция на основе директив препроцессора ifndef-define-endif называется «стражи включения» [*include guards*] и защищает от повторного включения заголовка в одну и ту же единицу трансляции.

Пример 6.2. Реализация модуля 1

```
// module_1.c
#include "module_1.h"
// Другие необходимые директивы include.
...
// Любые внутренние объявления и определения.
...
// Определения переменных и функций,
// объявленных в module_1.h.
...
```

Предположим, что модуль 2 использует модуль 1 в своей реализации. Тогда соответствующая единица трансляции может иметь следующее начало:

Пример 6.3. Реализация модуля 2

```
// module_2.c
#include "module_2.h"
#include "module_1.h" // подключить модуль 1
// Далее аналогично модулю 1.
...
```

Каждая единица трансляции компилируется независимо от прочих («раздельная компиляция»), затем все они связываются компоновщиком в единый файл машинного кода (исполняемый файл или двоичную библиотеку).

Реализация модуля может сразу предоставляться в скомпилированном виде (как объектный файл или двоичная библиотека), что дает возможность сокрытия исходного кода реализации от пользователей модуля.

Для того чтобы объявить в заголовочном файле переменную, не определяя ее, следует использовать ключевое слово **extern** (от *external linkage* — «внешнее связывание»):

Пример 6.4. Интерфейс модуля

```
// shared_var.h
#ifndef SHARED_VAR_H_INCLUDED
#define SHARED_VAR_H_INCLUDED
// Объявление общей переменной.
extern int shared_var;
// Не может содержать инициализатор!
#endif//SHARED_VAR_H_INCLUDED
```

Пример 6.5. Реализация модуля

```
// shared_var.c
#include "shared_var.h"
// Определение общей переменной.
int shared_var = 42; /* Эта переменная
  размещается в единице трансляции shared_var.c
  (в ее области памяти). */
```

Слово **extern** можно применить к прототипу функции, но обычно это не требуется, поскольку прототип функции и так по умолчанию подразумевает объявление функции, определенной где-то в другом месте и использующей внешнее связывание.

В C++ **extern** получило дополнительную функцию — объявление стандарта связывания (и, возможно, соглашения вызова), используемого для переменной или функции. Для этого после **extern** следует разместить строковую константу с именем

языка программирования, стандарт связывания которого следует применять к данной функции. Гарантируется поддержка двух вариантов: "C" и "C++". Наличие иных вариантов зависит от компилятора и платформы.

```
// Функция определена в библиотеке на языке C.
extern "C" float sqrtf(float);
```

Наиболее часто используется спецификатор **extern** "C" для импорта функций из двоичных библиотек, определенных на чистом C или имеющих стандартный двоичный интерфейс C[76]. Чтобы не писать каждый раз **extern**, объявления можно взять в фигурные скобки:

```
// sockets.h
#ifndef SOCKETS_H_INCLUDED
#define SOCKETS_H_INCLUDED
#ifdef __cplusplus // C++?
extern "C" {
#endif
int open_socket(int addr);
int close_socket(int socket);
int read_socket(int socket, void * buf, int size);
int write_socket(int socket, void * data, int size);
#ifdef __cplusplus
} // закрыть extern
#endif
#endif//SOCKETS_H_INCLUDED
```

Так как каждая единица трансляции требует компиляции и компоновки с прочими единицами трансляции, то могут потребоваться дополнительные усилия для правильной сборки кода, использующего библиотеки, предоставленные в исходном коде. Поэтому некоторые библиотеки, распространяемые в виде исходного кода на языке C++, состоят только из заголовочных файлов («header-only libraries»). Они не требуют отдель-

[76] Что широко поддерживается разными языками программирования как общий стандарт межъязыкового взаимодействия.

ной сборки и просты в использовании (обычно достаточно подключить тот или иной заголовочный файл с помощью include).

C++ позволяет определять функции и переменные прямо в заголовочном файле, при этом задачу размещения этих определений в объектных файлах можно переложить на компилятор.

Чтобы определить такую функцию или переменную, следует использовать ключевое слово **inline**:

```
// library.hpp, C++17
#ifndef LIBRARY_HPP_INCLUDED
#define LIBRARY_HPP_INCLUDED
inline int debug_calls = 0;
inline void debug_call() {
  #ifndef NDEBUG
  ++debug_calls;
  #endif
}
#endif//LIBRARY_HPP_INCLUDED
```

Если разные единицы трансляции включают library.hpp, то они получают доступ к одним и тем же переменной debug_calls и функции debug_call. Если разные единицы трансляции включают разнотипные одноименные inline-переменные или inline-функции с одной сигнатурой, то это влечет неопределенное поведение.

Пространство имен [*namespace*] — множество имен.

Цель введения пространств имен состоит в борьбе с *конфликтами имен* — той неприятной ситуацией, когда разные части программы или разные используемые программой библиотеки определяют разные сущности под одинаковыми именами.

Пространство имен может быть *именованным* (иметь имя) или *анонимным* (не иметь имени[77]).

[77] Компилятор может назначать каждому анонимному пространству имен уникальное имя. Обычно это имя не может быть доступно из программы, так как использует запрещенные в идентификаторах символы.

Можно сказать, что пространства имен дополняют возможности модульного программирования в C++. В то же время они ортогональны модулям (единицам трансляции): один модуль может использовать несколько пространств имен, одно пространство имен может содержать имена, определенные в разных модулях.

Синтаксис описания пространства имен выглядит следующим образом (в случае анонимного пространства имен имя-пространства-имен не указывается):

```
namespace имя-пространства-имен {
  // Объявления и определения,
  // помещаемые в данное пространство имен.
  ...
}
```

Строго говоря, вся эта конструкция целиком не является ни объявлением, ни определением. Аналогичных конструкций для одного именованного пространства имен может быть много (если имени нет, то каждый раз создается новое анонимное пространство имен), они могут быть собраны в одном файле или разбросаны по разным файлам. Каждая из них пополняет свое пространство имен перечисленными объявлениями и определениями. Можно сказать, что это просто набор объявлений и определений — синтаксис, позволяющий к каждому имени добавить префикс соответствующего пространства имен.

Конструкция вида (C++17):

```
namespace A::B::C {
  ...
}
```

эквивалентна набору вложенных пространств имен:

```
namespace A { namespace B { namespace C {
  ...
}}}
```

Обратиться к имени из пространства имен можно с помощью оператора ::, что мы уже видели раньше на примере пространства имен std.

Имена, определенные вне пространств имен, попадают в неименованное *глобальное пространство имен*. К ним можно обратиться явно, используя унарный оператор ::. Это может потребоваться в том случае, если внутреннее имя совпало с именем из глобального пространства имен и *затенило* [*shadowed*] его.

```
#include <iostream>
namespace my {
  namespace std { int cout = 0; }
  void foo() {
    // Внутреннее std затеняет внешнее.
    ::std::cout << std::cout;
  }
}
```

Если имена объявлены в некотором пространстве имен (например, в заголовочном файле), то их определения должны быть помещены в то же пространство имен. Это можно сделать так же, как и для объявлений — с помощью **namespace**, а можно — с помощью оператора ::, например:

```
// Объявление.
namespace my { int sqr(int); }
// Определение.
int my::sqr(int x) {
  return x * x;
}
```

C++ позволяет объявить синоним имени пространства имен, что используется для сокращения записи длинных вложенных имен:

```
namespace brief_name = some::longer::namespace_name;
```

Чтобы вообще не указывать пространство имен при обращении к имени, следует использовать ключевое слово **using**. Первый вариант («директива **using**») включает автоматический поиск во всем пространстве имен. Мы его уже использовали (для пространства имен std):

```
// Обращаться к данному пространству имен,
// если имя не найдено во вмещающем пространстве:
using namespace std;
```

Второй вариант («**using**-объявление») импортирует единственное имя, как если бы оно было объявлено в этом пространстве имен:

```
using std::cout; // взять из std только cout
cout << "Ok";
```

Анонимное пространство имен подразумевает директиву **using namespace** с назначенным ему именем сразу после закрывающей скобки. Таким образом, все имена из анонимного пространства имен видны коду, расположенному под ним.

Помимо введения имени в пространство имен, директива **using** может быть использована для объявления синонима имени типа:

```
using another_type_name = some_type;
```

После такой директивы имя another_type_name есть то же самое, что и some_type. Важно помнить, что это не определение нового типа, а всего лишь синоним, т. е. эти имена взаимозаменяемы.

Практический смысл синонима может заключаться в введении более удобной записи или в возможности замены часто используемого типа в одном месте (где объявлен синоним).

Одна и та же директива **using** может повторяться произвольное число раз.

Ключевое слово **static** (в C и при аналогичном применении в C++) позволяет явно указать *внутреннее связывание* [*internal linkage*] для переменной или функции. Соответственно, **static** и **extern** взаимно исключают друг друга. Переменные и функции с внутренним связыванием не доступны из других единиц трансляции.

В случае C **static** является единственным допустимым способом определить переменную или функцию прямо в заголовочном файле:

```
static int sqr(int x) {
  return x*x;
}
```

Каждая единица трансляции, включающая заголовок с таким определением, будет иметь *свою* копию функции sqr, не видимую из прочих единиц трансляции.

На практике это важно для переменных: **static**-переменные принадлежат только одной единице трансляции и не создают конфликтов имен при компоновке, так как компоновщик их не видит.

В C++ того же эффекта можно добиться при помощи анонимного пространства имен. Все переменные и функции, в нем определенные, имеют внутреннее связывание.

Резюмируя, можно сказать, что C++ предлагает следующие режимы связывания переменных и функций:

- Нет связывания: все переменные, определенные внутри функций (локальные переменные).
- Внутреннее связывание: функции и глобальные переменные, определенные со спецификатором **static** или в анонимном пространстве имен, а также глобальные константы (**const**[78]).
- Внешнее связывание: функции и глобальные переменные, определенные без **static** и, возможно, в именованном пространстве имен; функции и переменные, объявленные со спецификатором **extern**.

Если переменная объявлена с **extern** в теле функции, то это глобальная переменная с внешним связыванием.

Спецификатор **inline** не следует указывать вместе со **static** или **extern** (хотя это и не запрещено и не влияет на режим связывания). Все **inline** переменные и функции следует включать во все использующие их единицы трансляции как определения.

[78] В C иначе: **const** не подразумевает **static**.

Глобальные переменные инициализируются до начала исполнения функции main. Память, в которой размещаются глобальные переменные, предварительно инициализируется нулями. Поэтому глобальные переменные встроенных типов являются инициализированными даже в случае отсутствия инициализатора. Относительный порядок их инициализации определен только в рамках одной единицы трансляции (в порядке указания определений сверху вниз, уже после подстановки всех заголовков). Относительный порядок инициализации разных единиц трансляции не определен.

Задания для самопроверки

У Ответьте на следующие вопросы.

- Как указать компилятору, что значение переменной не должно изменяться после инициализации?
- Есть ли разница между следующими двумя определениями?

```
const int one = 1; // вариант 1
int const one = 1; // вариант 2
```

- Является ли модульное программирование императивным?
- На использование какой инструкции накладывает запрет структурное программирование?
- Какие файлы играют роль интерфейсной части модулей в языке C?
- Зачем нужны пространства имен в C++?
- Каким образом можно избежать повторного включения заголовочного файла в одну и ту же единицу трансляции?

- Пусть есть определение вида:

  ```
  int const ZERO = 0;
  ```

  Допустимо ли его разместить в заголовочном файле C++?

Глава 7

Элементарные вычисления

7.1. Числа с плавающей запятой

Для работы с вещественными числами могут использоваться различные способы покрытия вещественной оси набором представимых чисел. Как правило, они ограничиваются некоторым подмножеством рациональных чисел:

- дроби вида $\frac{m}{n}$, где m и n — целые числа. Представление есть пара чисел (m, n);
- *числа с фиксированной запятой*[79] [*fixed point numbers*]: в качестве представления используется целое число m, числовым значением является дробь вида $m \cdot r^{-p}$, где r и p — заранее заданные константы. Пример: пусть $r = 10$, $p = 4$, тогда $m = 125\,501$ представляет число 12.5501, а $m = 1$ представляет число 0.0001;

[79] Вместо слова *запятая* часто используется слово *точка*, так как в англоязычных странах и в языках программирования для разделения целой и дробной части числа традиционно используется точка. Поэтому оба этих варианта можно считать синонимами.

- *числа с плавающей запятой* [*floating point numbers*]: пара целых чисел (m, p) задает дробь вида $m \cdot r^p$, где только *основание* r является заранее заданной константой. Таким образом, позиция запятой (**порядок**) p (в r-ичной системе счисления) указана в самом представлении числа.

Сравнительно с числами с фиксированной запятой числа с плавающей запятой, имеющие тот же размер представления, позволяют существенно расширить покрываемый диапазон вещественной оси, однако при этом теряется однородность покрытия: чем дальше от нуля, тем больше шаг между соседними представимыми числами.

Основным стандартом, задающим форматы чисел с плавающей запятой и правила работы с ними, является стандарт IEEE-754 (в последней редакции известный также под обозначением ISO/IEC/IEEE 60559:2011). Его поддерживают большинство современных процессоров, умеющих работать с числами с плавающей запятой.

IEEE-754 задает двоичное представление числа, состоящее из трех частей: *знака* [*sign*], *порядка* [*exponent*] и *множителя* [*significand*[80]] (см. таблицу). Распространено также употребление слов «экспонента» и «мантисса» для обозначения порядка и множителя соответственно.

1. Знак s	2. Порядок E	3. Множитель M
1 разряд	e разрядов	m разрядов
старший бит	биты $(m+e-1)\ldots m$	биты $(m-1)\ldots 0$

Двоичное представление числа в IEEE-754

Необходимо отметить, что в данном представлении значение соответствующих бит-строк, взятых как целые E и M, не дает непосредственно $M \cdot r^E$, дело обстоит несколько сложнее.

[80] Еще синонимы данного термина: *показатель*, *коэффициент* [*coefficient*].

Любое двоичное представление относится к одному из пяти возможных классов:

1. *Нуль* [*zero*]: значение 0.

2. *Субнормальное число*[81] [*subnormal number*]: значения, близкие нулю по абсолютной величине; равномерно заполняют пространство между нулем и ближайшими (слева и справа от нуля) нормализованными числами.

3. *Нормальное* или *нормализованное число* [*normal number*]: «обычные» значения.

4. *Бесконечность* [*infinity*]: два специальных условных значения $-\infty$ и $+\infty$.

5. *Нечисло* [*not-a-number, NaN*]: коды ошибок, возвращаемые при попытке выполнять недопустимую операцию, например, поделить нуль на нуль.

IEEE-754 предлагает два набора форматов: двоичные $r = 2$ и десятичные $r = 10$. Аппаратно практически везде поддерживаются только двоичные форматы, и далее мы будем предполагать $r = 2$.

Порядок представлен в сдвинутом коде со сдвигом b, где

$$b = 2^{e-1} - 1.$$

Максимальное и минимальное значения E, как целого числа без знака, интерпретируются особым образом (см. таблицу, M подставляется в запись двоичной дроби как бит-строка).

Под нормализацией числа понимается его домножение на такую степень двойки, что слева от точки получается единица. Субнормальные числа требуют непредставимых значений экспоненты для нормализации.

[81] Также может называться *денормализованным числом* [*denormal number*].

Класс значения	E	M	Значение числа
нуль	0	0	$(-1)^s \cdot 0$
субнормальное	0	$\neq 0$	$(-1)^s \cdot 0.M \cdot 2^{1-b}$
нормальное	$1 \ldots 2b$	любое	$(-1)^s \cdot 1.M \cdot 2^{E-b}$
бесконечность	$2b+1$	0	$(-1)^s \cdot \infty$
нечисло	$2b+1$	$\neq 0$	код ошибки

Интерпретация представления числа с плавающей точкой

Замечания

• Субнормальные числа заполняют окрестность нуля с постоянным шагом.

• Наличие двух нулей (отрицательного и положительного) иногда полезно: так, при получении очень малых величин, округлившихся к нулю, сохранится знак (с какой стороны от нуля).

• Тем не менее оба нуля считаются равными друг другу. Кроме того, полагается $\sqrt{-0} = -0$, $\ln(-0) = -\infty$, $(\pm 0)^{\pm 0} = 1$.

• Бесконечность можно получить, например, поделив ненулевое число на нуль, или любым другим способом, при котором результат вычислений пусть даже и конечен, но слишком велик по абсолютной величине, чтобы быть представленным в используемом формате.

• Так как нулей два, то, например, $(-1)/(-0) = +\infty$.

• Нечисло можно получить, если, например, вычесть бесконечность из бесконечности или попробовать вычислить квадратный корень отрицательного числа (т. е. когда результат операции не определен, кроме некоторых специальных случаев вроде значения «бесконечность» и упомянутого выше 0^0).

• Арифметические операции и большинство математических функций возвращают нечисла, если хотя бы один аргумент или операнд является нечислом. Поэтому появившееся в одном месте нечисло может породить лавину нечисел в последующих вычислениях. Эта ситуация называется «заражение нечислами» [*NaN plagueing*].

• Для нечисел операторы сравнения всегда возвращают ложь (нечисло не считается равным даже само себе).

• Как правило, процессоры гораздо медленнее выполняют арифметические операции с субнормальными числами, бесконечностями и нечислами, чем с нормальными числами. Это может вызывать резкое падение производительности в некоторых случаях.

• Сумма и произведение чисел с плавающей запятой, строго говоря, не являются ассоциативными операциями: в общем случае $a + (b + c) \neq (a + b) + c$.

Двоичные форматы IEEE-754

Опишем четыре основных двоичных формата чисел с плавающей запятой, предлагаемых стандартом IEEE-754. Название формата включает в себя размер его представления в битах (16, 32, 64 и 128):

- **Binary16**, известный также как *число половинной точности* [*half precision number*], популярен в компьютерной графике и вычислениях, использующих искусственные нейронные сети;

- **Binary32**, известный также как *число одинарной точности* [*single precision number*], широко используется для вычислений, не требующих высокой точности (например, геометрические данные в компьютерной графике). Тип **float** в большинстве реализаций C++ реализует именно этот формат;

- **Binary64**, известный также как *число двойной точности* [*double precision number*], широко используется в научных и инженерных вычислениях. В большинстве реализаций C++ тип данных **double** реализует этот формат;

- **Binary128**, известный также как *число четверной точности* [*quadruple precision number*], используется редко ввиду малой доступности аппаратных реализаций и низкой скорости работы программных реализаций. В некоторых реализациях C++ может быть доступен как _Quad, __float128 или **long double**.

Размеры в битах порядка и множителя для перечисленных форматов указаны в таблице.

Формат	e	m	Сдвиг b
Binary16	5	10	15
Binary32	8	23	127
Binary64	11	52	1023
Binary128	15	112	16383

Двоичные форматы IEEE-754: размеры

В следующей таблице описаны точность и диапазоны представлений чисел, обеспечиваемые данными форматами.

Формат	d	ε	min суб.	min норм.	max норм.
Binary16	5	$9.77 \cdot 10^{-4}$	$5.96 \cdot 10^{-8}$	$6.1 \cdot 10^{-5}$	65504
Binary32	9	$1.19 \cdot 10^{-7}$	$1.4 \cdot 10^{-45}$	$1.18 \cdot 10^{-38}$	$3.4 \cdot 10^{38}$
Binary64	17	$2.22 \cdot 10^{-16}$	$4.94 \cdot 10^{-324}$	$2.2 \cdot 10^{-308}$	$1.8 \cdot 10^{308}$
Binary128	36	$1.93 \cdot 10^{-34}$	$6.48 \cdot 10^{-4966}$	$3.4 \cdot 10^{-4932}$	$1.2 \cdot 10^{4932}$

Двоичные форматы IEEE-754: точность

Число ε — наименьшее положительное представимое число такое, что $1 + \varepsilon \neq 1$ в данном формате. В колонках «min

суб.», «min норм.» и «max норм.» указаны минимумы и максимумы субнормальных и нормальных чисел. В качестве числа d в таблице указано минимальное число десятичных знаков после запятой, которого достаточно для сохранения двоичного представления числа при преобразовании в десятичную форму и обратно. Необходимо помнить, что многие «обычные» десятичные дроби являются бесконечными в двоичной системе: например, 0.1 непредставимо точно в указанных форматах, так как содержит делитель 5, взаимопростой с основанием $r = 2$. Двоичные дроби с конечным числом знаков представимы точно десятичными дробями с конечным числом знаков, однако могут потребовать сотни десятичных цифр для точной записи даже в формате Binary32.

Заметим, что Binary32 не позволяет гарантированно сохранить точное значение 32-битного целого, так как множитель (вместе с целой частью) имеет длину лишь 24 бита. Аналогичные рассуждения справедливы и для других форматов: для гарантированно точного представления целого числа нам нужно сделать «шаг назад» в разрядности: Binary16 способен хранить 8-битные целые, Binary32 — 16-битные, Binary64 — 32-битные.

Режимы округления

Стандарт IEEE-754 предусматривает наличие пяти режимов округления, применяемых к результату операции для получения итогового представления:

- к ближайшему, половинки к четному [*to nearest, ties to even*] (в таблице ниже «б/ч»): берется ближайшее представимое число, а если таковых два, то берется «четное» (младший бит нулевой). Данный способ известен также как «банковское округление» и позволяет минимизировать накапливаемую погрешность. Обычно по умолчанию используется именно этот режим;

- к ближайшему, половинки от нуля [*to nearest, away from zero*] (в таблице «б/от 0»): берется ближайшее предста-

вимое число, а если таковых два, то берется то из них, которое дальше от нуля (классический «школьный» способ);

- к нулю [*toward zero*], отбрасывание [*truncation*] (в таблице «к 0»): обнуляем отбрасываемые разряды;
- к $+\infty$ [*to positive infinity*], «наверх» [*upwards*], «потолок» [*ceiling*] (в таблице «к $+\infty$»): из двух представимых чисел, между которыми лежит округляемое значение, выбирается наибольшее;
- к $-\infty$ [*to negative infinity*], «вниз» [*downwards*], «пол» [*floor*] (в таблице «к $-\infty$»): из двух представимых чисел, между которыми лежит округляемое значение, выбирается наименьшее.

Их отличие друг от друга можно проиллюстрировать на примере округления до целых чисел с помощью следующей таблицы.

Режим	-2.7	-2.5	-2.2	-1.5	1.5	2.2	2.5	2.7
б/ч	-3.0	-2.0	-2.0	-2.0	2.0	2.0	2.0	3.0
б/от 0	-3.0	-3.0	-2.0	-2.0	2.0	2.0	3.0	3.0
к 0	-2.0	-2.0	-2.0	-1.0	1.0	2.0	2.0	2.0
к $+\infty$	-2.0	-2.0	-2.0	-1.0	2.0	3.0	3.0	3.0
к $-\infty$	-3.0	-3.0	-3.0	-2.0	1.0	2.0	2.0	2.0

Режимы округления IEEE-754

В контексте C++

Запись констант с плавающей точкой в исходном коде C++ обязана содержать точку или символ, отделяющий порядок от множителя (e, E или p, P), в противном случае они трактуются как целые числа. Целая либо дробная часть числа при

этом может быть опущена, если она нулевая. Например, 10^{10} как число с плавающей запятой может быть оформлено как 1e10, 10'000'000'000., 10.0e+9 и т. д. Все эти представления равнозначны.

Чтобы задать точное двоичное представление константы, следует воспользоваться префиксом 0x, 0X (C99, C++11). В этом случае дробь записывается в шестнадцатеричной системе счисления и позволяет указать множитель итогового представления с точностью до бита. Показатель записывается после буквы p или P в десятичной форме, но задает степень основания r (в двоичных форматах $r = 2$), например, 0xA.BCDp+4 равно $(10 + 11 \cdot 16^{-1} + 12 \cdot 16^{-2} + 13 \cdot 16^{-3}) \cdot 2^4 = 171.80078125$.

Одна из типичных ошибок новичков — использование целочисленных констант там, где необходимы константы в плавающей точке.

Пример 7.1. Неверный способ вычислить кубический корень

```
double broken_cubic_root(double x) {
  return pow(x, 1/3); // возводит в степень 0!
}
```

В примере выше 1 и 3 — целые числа, поэтому к ним применяется целочисленное деление (с отбрасыванием остатка), т. е. 1/3 дает 0. В итоге, функция broken_cubic_root не вычисляет кубический корень. Правильным вариантом будет следующий код (также с учетом знака, так как стандартная функция возведения в степень не принимает отрицательное основание и нецелый показатель):

Пример 7.2. Способ вычислить кубический корень

```
double cubic_root(double x) {
  return x < 0.0?
      -pow(-x, 1./3)
    :  pow( x, 1./3);
}
```

В современном C++, впрочем, имеется стандартная функция для вычисления кубического корня cbrt. При ее наличии следует использовать ее вместо варианта на основе pow.

Языки C и C++ предоставляют три встроенных типа для работы с числами с плавающей точкой:

- **float** обычно соответствует IEEE-754 Binary32. Литералы этого типа задаются с помощью суффиксов f, F, например, 3.14f, 2E−1F;

- **double** обычно соответствует IEEE-754 Binary64. Данный тип используется по умолчанию для литералов без суффиксов. Также рекомендуется использовать данный тип для вычислений, если нет причин выбрать другой тип;

- **long double** на платформе x86 нередко отождествляется с 80-битным форматом сопроцессора Intel 8087 (использование которого на платформе x86-64 объявлено устаревшим), в то время как, например, Microsoft Visual C++ использует представление, идентичное типу **double**, а некоторые другие компиляторы могут использовать IEEE-754 Binary128. Литералы данного типа задаются с помощью суффиксов l, L, например, 1e−1000L.

Стандарт C++ гарантирует лишь отношение множеств значений вида **float** $\subseteq$ **double** $\subseteq$ **long double**.

Часть характеристик форматов с плавающей запятой, используемых конкретным компилятором, можно получить, подключив заголовочный файл **cfloat**, где определен ряд констант. Ниже $*$ соответствует FLT для типа **float**, DBL для **double** и LDBL для **long double**:

- FLT_RADIX основание системы счисления r (обычно 2);

- FLT_ROUNDS режим округления по умолчанию;

- $*$_MANT_DIG разряды множителя (включая бит целой части) $(m + 1)$;

• *_DIG достаточное число десятичных знаков после запятой d;

• *_MIN_EXP минимальная смещенная экспонента для нормальных чисел;

• *_MIN_10_EXP минимальная десятичная экспонента для нормальных чисел;

• *_MAX_EXP максимальная смещенная экспонента для нормальных чисел;

• *_MAX_10_EXP максимальная десятичная экспонента;

• *_MIN наименьшее положительное нормальное число;

• *_TRUE_MIN наименьшее положительное (субнормальное) число;

• *_MAX наибольшее конечное представимое число;

• *_EPSILON наименьшее положительное представимое число ε такое, что $1 + \varepsilon \neq 1$ в заданном формате.

7.2. Математические функции

Название данного раздела условно — речь идет о стандартных функциях, предназначенных для работы с числами с плавающей запятой.

Заголовочный файл **cmath** предоставляет ряд определений математических функций, вычисляемых приближенно в плавающей точке, но для простоты обозначаемых как оперирующие действительными числами. Здесь приводятся сведения по определениям для C++, принимающим разнотипные значения (**float**, **double** и **long double**). Если предел той или иной функции при стремлении к нулю (с той или иной стороны) или бесконечностям определен, то он берется в качестве значения этой

функции в соответствующих предельных точках (поэтому в колонке Ran стоят отрезки, например, включающие бесконечности). Итак, для каждой функции указана область определения Dom и область значений Ran.

Функция	Смысл	Dom	Ran
$\mathrm{abs}(x)$, $\mathrm{fabs}(x)$	модуль $\lvert x\rvert$	$\mathbb{R}$	$[0,\infty]$
$\mathrm{signbit}(x)$	бит знака	$\mathbb{R}$	**bool**
$\mathrm{copysign}(x,\,y)$	$\lvert x\rvert\operatorname{sgn} y$	$\mathbb{R}^2$	$\mathbb{R}$
$\mathrm{fmax}(x,\,y)$	$\max\{x,y\}$	$\mathbb{R}^2$	$\mathbb{R}$
$\mathrm{fmin}(x,\,y)$	$\min\{x,y\}$	$\mathbb{R}^2$	$\mathbb{R}$
$\mathrm{fmod}(x,\,y)$	остаток	$\lvert x\rvert<\infty$, $y\neq 0$	$[0,x)$
$\mathrm{fma}(x,\,y,\,z)$	$xy+z$	$\mathbb{R}^3$	$\mathbb{R}$

Функции общего назначения

Функция abs определена и для целых (возвращает целое) и для дробных типов. Функция fabs определена для дробных типов и в случае передачи аргумента целочисленного типа приводит его к **double** (это касается и многих других функций, не имеющих особых определений для целочисленных типов).

Функция signbit извлекает бит знака и возвращает **false**, если он нулевой (в частности, для положительных чисел), и **true**, если он ненулевой (в частности, для отрицательных чисел).

Функции fmax и fmin возвращают другой аргумент, если один из аргументов — нечисло.

Функция fmod является аналогом оператора % для дробных значений: $\mathrm{fmod}(x,y) \triangleq x - y\left\lfloor\frac{x}{y}\right\rfloor$. Результат имеет знак аргумента x.

Функция fma (*слитое умножение-сложение* [*fused multiply-add*]) избегает промежуточного округления, чем обеспечивается бóльшая точность сравнительно с арифметической формулой, а также отсутствие промежуточного переполнения, если конечный результат представим как число. Кроме того, многие современные процессоры поддерживают данное действие аппаратно, за счет чего данная операция может выполнять-

ся *быстрее*, чем две арифметических операции (умножение и сложение).

Функция	Округление	Dom	Ran
ceil(x)	$\lceil x \rceil$	$\mathbb{R}$	$\mathbb{Z} \cap \mathbb{R}$
floor(x)	$\lfloor x \rfloor$	$\mathbb{R}$	$\mathbb{Z} \cap \mathbb{R}$
trunc(x)	к 0	$\mathbb{R}$	$\mathbb{Z} \cap \mathbb{R}$
round(x)	к б/от 0	$\mathbb{R}$	$\mathbb{Z} \cap \mathbb{R}$
rint(x)	тек. режим	$\mathbb{R}$	$\mathbb{Z} \cap \mathbb{R}$
nearbyint(x)	тек. режим	$\mathbb{R}$	$\mathbb{Z} \cap \mathbb{R}$

Функции округления до целых (1)

Функции ceil, floor, trunc и round реализуют разные варианты округления аргумента x до целого числа n: ceil — к $+\infty$, floor — к $-\infty$, trunc отбрасывает дробную часть, округляя в направлении нуля, round — к ближайшему, $|n-x| \leqslant 0.5$, «половинки» в сторону от нуля. Все эти функции возвращают число того же типа, что и аргумент (в плавающей точке). Функция rint использует *текущий режим округления*, который обычно по умолчанию соответствует округлению к ближайшему, «половинки» к четным:

```
std::cout << std::round(0.5) << '\n'; // 1
std::cout << std::round(1.5) << '\n'; // 2
std::cout << std::rint(0.5) << '\n';  // 0 (четное!)
std::cout << std::rint(1.5) << '\n';  // 2
```

Единственное отличие nearbyint от rint в том, что rint может сигнализировать об ошибке «потеря точности» (FE_INEXACT), если аргумент не равен результату (не был целым).

Функции lround, llround, lrint и llrint возвращают значения целочисленных типов. В случае, если результат непредставим в виде значения соответствующего типа или является бесконечностью или нечислом, функции сигнализируют об ошибке «неверный аргумент» (FE_INVALID) и возвращают значение, определяемое реализацией стандартной библиотеки.

Функция	Округление	Dom	Ran
lround(x)	к б/от 0	$\mathbb{R}$	**long int**
lrint(x)	тек. режим	$\mathbb{R}$	**long int**
llround(x)	к б/от 0	$\mathbb{R}$	**long long int**
llrint(x)	тек. режим	$\mathbb{R}$	**long long int**

Функции округления до целых (2)

Функция	Смысл	Dom	Ran
sqrt(x)	$\sqrt{x}$	$[0, \infty]$	$[0, \infty]$
cbrt(x)	$\sqrt[3]{x}$	$\mathbb{R}$	$\mathbb{R}$
exp(x)	e^x	$\mathbb{R}$	$[0, \infty]$
exp2(x)	2^x	$\mathbb{R}$	$[0, \infty]$
expm1(x)	$e^x - 1$	$\mathbb{R}$	$[-1, \infty]$
log(x)	$\ln x$	$[0, \infty]$	$\mathbb{R}$
log2(x)	$\log_2 x$	$[0, \infty]$	$\mathbb{R}$
log10(x)	$\log_{10} x$	$[0, \infty]$	$\mathbb{R}$
log1p(x)	$\ln(x + 1)$	$[-1, \infty]$	$\mathbb{R}$
pow(x, y)	x^y	см. ниже	$\mathbb{R}$
hypot(x, y)	$\sqrt{x^2 + y^2}$	$\mathbb{R}^2$	$[0, \infty]$
hypot(x, y, z)	$\sqrt{x^2 + y^2 + z^2}$	$\mathbb{R}^3$	$[0, \infty]$

Степени и логарифмы

В таблице $e \approx 2.718281828459045$ есть основание натурального логарифма (ln).

С помощью функции hypot можно вычислить евклидову длину вектора из $\mathbb{R}^2$ или из $\mathbb{R}^3$ (C++17). Функция возвращает корректный результат, если он представим.

Функции expm1 и log1p обеспечивают высокую точность вычисления при $x \approx 0$.

Функция pow(x, y) предназначена для возведения в степень. В случае $x > 0$ допустимы любые показатели y, для вычисления применяется тождество $x^y = 2^{y \log_2 x}$ или аналогичное. В случае $x < 0$ допустимы только целые y, четность y определяет знак результата. Если в качестве y передать значе-

ние типа **int**, то результат будет иметь тип **double** независимо от типа значения x (C++11).

При вычислении малых целых степеней рекомендуется использовать умножение (и деление) вместо pow:

`x*x` вместо `pow(x, 2)`, `x*x*x` вместо `pow(x, 3)`,
`1./(x*x)` вместо `pow(x, -2)` и т. д.

При вычислении квадратного корня вызов $\mathrm{sqrt}(x)$ лучше вызова $\mathrm{pow}(x, 0.5)$. При вычислении кубического корня вызов $\mathrm{cbrt}(x)$ лучше вызова $\mathrm{pow}(x, 1./3)$, как минимум по точности, и корректно обрабатывает случай $x < 0$.

Функция	Смысл	Dom	Ran
$\sin(x)$	$\sin x$	$\mathbb{R}$	$[-1, 1]$
$\cos(x)$	$\cos x$	$\mathbb{R}$	$[-1, 1]$
$\tan(x)$	$\mathrm{tg}\, x$	$x \neq \frac{\pi n}{2}, n \in \mathbb{Z}$	$\mathbb{R}$
$\mathrm{asin}(x)$	$\arcsin x$	$x \in [-1, 1]$	$[-\frac{\pi}{2}, \frac{\pi}{2}]$
$\mathrm{acos}(x)$	$\arccos x$	$x \in [-1, 1]$	$[0, \pi]$
$\mathrm{atan}(x)$	$\mathrm{arctg}\, x$	$\mathbb{R}$	$[-\frac{\pi}{2}, \frac{\pi}{2}]$
$\mathrm{atan2}(y, x)$	$(\mathrm{arctg}\, \frac{y}{x})$	$\mathbb{R}^2$	$[-\pi, \pi]$

Тригонометрические функции

Функция $\mathrm{atan2}(y, x)$ вычисляет $\arg(x + yi)$ или, что то же самое, угол между векторами $(1, 0)^\top$ и $(x, y)^\top$. Как известно, тангенс этого угла равен $\frac{y}{x}$. Однако частное не сохраняет информации о знаках x и y и не позволяет определить квадрант, в котором находится точка (x, y). Функция atan2 решает эту задачу, принимая координаты по отдельности. В рамках IEEE-754 функция atan2 определена для всех возможных значений, включая оба нуля и бесконечности.

Функция tan округляется до бесконечности вблизи значений $\frac{\pi n}{2}, n \in \mathbb{Z}$. На практике точность вычисления данной функции может быть неудовлетворительной из-за ее очень быстрого роста и приводить к значительной погрешности вычислений по формулам, содержащим тангенс.

Функция	Смысл	Dom	Ran
$\sinh(x)$	$\mathrm{sh}\, x$	$\mathbb{R}$	$\mathbb{R}$
$\cosh(x)$	$\mathrm{ch}\, x$	$\mathbb{R}$	$[1, \infty]$
$\tanh(x)$	$\mathrm{th}\, x$	$\mathbb{R}$	$[-1, 1]$
$\mathrm{asinh}(x)$	$\mathrm{Arsh}\, x$	$\mathbb{R}$	$\mathbb{R}$
$\mathrm{acosh}(x)$	$\mathrm{Arch}\, x$	$x \in [1, \infty]$	$[0, \infty]$
$\mathrm{atanh}(x)$	$\mathrm{Arth}\, x$	$x \in [-1, 1]$	$\mathbb{R}$

Гиперболические функции

Наконец, есть набор функций, выполняющих классификацию и сравнение значений с плавающей запятой.

Во-первых, это функция fpclassify , возвращающая класс значений, которому принадлежит ее аргумент в виде константы (некое целое число):

- FP_ZERO для нулей;
- FP_SUBNORMAL для субнормальных чисел;
- FP_NORMAL для нормальных чисел;
- FP_INFINITE для бесконечностей;
- FP_NAN для нечисел.

Во-вторых, это функции, возвращающие булевское значение и проверяющие принадлежность значения тому или иному классу:

- isnormal(x) эквивалентно fpclassify (x) == FP_NORMAL;
- isinf(x) эквивалентно fpclassify (x) == FP_INFINITE;
- isnan(x) эквивалентно fpclassify (x) == FP_NAN;
- isfinite(x) эквивалентно !(isinf (x) || isnan(x)).

В-третьих, это функции, выполняющие сравнение значений по величине. В отличие от встроенных операторов сравнения, данные функции не сигнализируют об ошибке, если один или оба их аргумента — нечисла (первые четыре функции в этом случае возвращают ложь):

- isless(x, y): x «меньше» y;
- islessequal(x, y): x «меньше или равно» y;
- isgreater(x, y): x «больше» y;
- isgreaterequal(x, y): x «больше или равно» y;
- isunordered(x, y): x «не упорядочено относительно» y, возвращает истину только в том случае, если x или y — нечисло.

Кроме того, файл **cmath** определяет константы:

- INFINITY — условное значение $+\infty$, тип **float** (соответственно, $-\infty$ можно записать как –INFINITY);
- NAN — некое нечисло.

7.3. Настройки арифметики чисел с плавающей запятой

Стандартные средства управления режимом округления и реакцией на исключительные ситуации (ошибки, приводящие к появлению нечисел, переполнения и округления со значительной потерей точности) предоставлены в заголовочном файле **cfenv**.

Подключив данный заголовочный файл, можно управлять текущим режимом округления. Возможен выбор из четырех стандартных (по стандарту C) режимов:

- FE_DOWNWARD — округление к $-\infty$;

- FE_TONEAREST — округление к ближайшему (стандарт не указывает, какой вариант округления «половинок» будет использоваться в данном режиме);
- FE_TOWARDZERO — округление к нулю;
- FE_UPWARD — округление к $+\infty$.

Данные целочисленные константы определены, если платформа поддерживает соответствующие режимы. Платформа может также определять дополнительные режимы.

Для управления текущим режимом округления предусмотрены функции:

- **int** fegetround() — функция возвращает текущий режим округления.
- **int** fesetround(**int** mode) — функция устанавливает новый режим округления. Возвращает 0 в случае успеха. В случае ошибки возвращает ненулевое значение.

Режим округления не влияет на округление до целых через приведение типа, работу функций ceil, floor, trunc, round, lround и llround.

Задания для самопроверки

У Ответьте на следующие вопросы.

- Является ли расстояние между любыми соседними представимыми в форматах с плавающей запятой числами одинаковым?
- Может ли повлечь потерю разрядов домножение числа на степень двойки?
- Пусть число в формате IEEE-754 Binary32 имеет двоичное представление $80\,00\,00\,00_{16}$. Каково его числовое значение?

- Пусть число в формате IEEE-754 Binary32 имеет двоичное представление $3F\,80\,00\,00_{16}$. Каково его числовое значение?

- Что выведет следующий код?

```
#include <cmath>
#include <iostream>
int main() {
  using namespace std;
  if (signbit(0) == signbit(-0))
    cout << '1';
  if (signbit(0.) == signbit(-0.))
    cout << '2';
}
```

- В чем разница между округлениями: **long**(x) и lrint(x)?

- Назовите две причины, по которым вызов isnan(x) лучше проверки через сравнение на равенство (!(x == x))?

- Является ли следующая функция тождественным преобразованием?

```
double foo(double x) {
  return ln(exp(x));
}
```

- Является ли множество значений типа float линейно упорядоченным?

У Пусть тип **float** реализует представление чисел в формате Binary32. Приведите пример значений a, b, c типа **float** таких, что $(a + b) + c \neq a + (b + c)$.

У Напишите функцию nroot, вычисляющую корень заданной степени. Пусть данная функция использует функции sqrt для случая n = 2, cbrt для случая n = 3, pow для n > 3.

```
double nroot(double x, int n);
```

Функция nroot должна корректно обрабатывать случай n — нечетное, $x < 0$.

Что делать, если $n < 0$?

Что делать, если $n = 0$?

У Стандартная функция nextafter(x, y) возвращает соседнее с x значение, ближайшее к y (или x, если $x == y$). Используя данную функцию, напишите цикл, выводящий (в «столбик») все субнормальные и нормальные числа, представимые как **float**.

У Представим, что требуется особый формат Binary8, аналогичный двоичным форматам IEEE-754, но имеющий представление шириной 8 бит. Предложите характеристики этого формата: число бит e и число бит m. Вычислите характеристики b, d, ε, минимальное положительное субнормальное число, минимальное положительное нормальное число, максимальное нормальное число.

У Реализуйте функции преобразования представлений между форматами Bfloat16 и Binary32. Числа в формате Bfloat16 имеют размер 16 бит и отличаются от Binary32, усеченной до 7 бит длиной множителя, имея то же самое представление порядка.

Глава 8

Указатели и массивы

8.1. Указатели

Ранее мы рассмотрели ссылки и ссылочный тип. Однако, до того как ссылки были введены в C++, в том же качестве и в языке C, и еще раньше в BCPL и языках низкого уровня можно было использовать *указатели*. Достаточно естественной представляется возможность интерпретировать значение ячейки памяти как адрес другой ячейки памяти и, вообще, обращаться с адресом как с целым числом.

Указатель [*pointer*] — переменная, значением которой является адрес.

Разыменованием указателя [*pointer dereferencing*] называют обращение к значению по адресу, хранимому указателем.

Указатель добавляет уровень косвенности и позволяет через одну переменную (себя) получить доступ к любой другой переменной некоторого типа (если изменить записанный в него адрес).

В C и C++ указатель определяется с помощью символа * после типа данного, на которое этот указатель будет указывать:

```
// Неинициализированный указатель на int!
int * p;
```

Здесь переменная p имеет тип **int***, множеством значений которого являются адреса ячеек типа **int**. Попытка обратиться по неинициализированному указателю, естественно, влечет неопределенное поведение.

Так же, как и & при определении ссылки, символ * «прилипает» к имени переменной:

```
int * p, * q, n;
// p и q имеют тип int*
// n имеет тип int
```

В отличие от ссылок, указатели поддерживают особое значение **нулевой указатель** [*null pointer*], которое не может быть адресом какой-либо ячейки памяти, выделяемой компилятором или средствами стандартной библиотеки. Это значение используется как признак того, что указатель ни на что не указывает. Инициализировать указатель как нулевой можно с помощью ключевого слова **nullptr**:

```
// Инициализированный указатель:
int * p = nullptr;
```

Значение **nullptr** является единственным значением типа std::nullptr_t (объявлен в **cstddef**), для которого определено неявное приведение к типу любого указателя. Это значение можно использовать в сравнениях или передавать в качестве значения указателя в функции, принимающие указатели.

При использовании в качестве условия или явном приведении к **bool** нулевой указатель приводится к **false**, а любой другой (ненулевой) — к **true**. Разыменование нулевого указателя влечет неопределенное поведение.

Ограничения, накладываемые на ссылки по сравнению с указателями, позволяют, с одной стороны, защитить программиста от ряда ошибок и, с другой стороны, открывают ряд возможностей оптимизации кода для компилятора. Ссылки используются там, где нет нужды в «полноценных» указателях или есть желание не перегружать код взятиями адреса и разыменованиями. Указатели обычно используются либо из-за

необходимости реализовать отложенную привязку или изменение привязки, либо из-за желания использовать нулевой указатель, либо из-за желания явно показать в коде, что осуществляется передача переменной, а не значения, и эта переменная может быть изменена.

Наличие нулевого указателя позволяет, например, возвращать указатель на искомый объект и в том случае, когда ничего не было найдено. Просто в такой ситуации возвращаем нулевой указатель, а принимающая сторона должна быть готова к этому. Отсутствие необходимых проверок указателей на равенство нулю является одной из традиционных ошибок.

Указатели — обычные переменные. Указатели не «делают вид», что они — те значения в памяти, к которым они привязаны. Чтобы получить указатель на переменную, нужно явно взять ее адрес с помощью оператора &. Чтобы обратиться к переменной, на которую указывает указатель, требуется явно разыменовать его с помощью оператора *. Указатели можно сравнивать друг с другом на равенство. Указатели равны, если указывают на один и тот же объект, и не равны в противном случае.

```
int n = 0;
int * r = &n; // теперь r — указатель на n
n = 10;
cout << *r << '\n';          // 10
*r = 20;       // изменить n через указатель
cout << n << '\n';           // 20
cout << (&n == r) << '\n'; // 1
int m = 0;
r = &m;          // привязать к другой переменной
*r = 3;
cout << n << ", " << m << '\n'; // 20, 3
cout << (&n == r) << '\n';       // 0
```

В данном примере переменные n и m обязательно имеют разные адреса: &n == &m всегда ложно.

Так же, как и в случае ссылок, можно использовать ключевое слово **const**, чтобы создать указатель на константу, запре-

щая изменять переменную через него. Сам указатель при этом изменять можно. Указатели на константы могут быть привязаны к обычным переменным, но не наоборот. (Неявное приведение типа может лишь добавлять константность, но не удалять ее.) Всего возможно четыре варианта сочетаний **const** и $*$:

```
int x = 0, y = 1;
// Указатель на x «только для чтения»:
int const * p1 = &x;
y = *p1;   // можно
*p1 = 10;  // ошибка компиляции: *p1 — константа!
p1 = &y;   // можно: сам указатель p1 не константа!

// Теперь константа — сам указатель:
int * const p2 = &x;
p1 = p2;   // можно
y = *p2;   // можно
*p2 = 10;  // тоже можно!
p2 = &y;   // ошибка компиляции: p2 — константа!

// Указатель-константа указывает на константу:
int const * const p3 = &x;
y = *p3;   // можно
p1 = p3;   // можно
*p3 = 10;  // ошибка компиляции: *p3 — константа!
p3 = &y;   // ошибка компиляции: p3 — константа!
```

Разыменование указателя возвращает ссылку.

Выражение	Тип
&x	**int***
p1	**int const***
*p1	**int const&**
p2	**int* const**
*p2	**int&**
p3	**int const* const**
*p3	**int const&**

Допустимо создавать ссылки на указатели (как и на любые прочие переменные). Указатель на ссылку невозможен.

```
int * p = nullptr;
int * & r = p;   // можно: r — синоним p
int n = 0;
int & e = n;     // &e имеет тип int*
int & * q = &e;  // ошибка компиляции!
```

Указатели можно передавать в функции и возвращать из функций как любые простые значения. Например:

```
// Передача «по указателю».
int * max_byptr(int * a, int * b) {
  return *a < *b? b: a;
}

int main() {
  int x = 0, y = 0;
  cin >> x >> y;
  *max_byptr(&x, &y) = 42;
  cout << "x = " << x << "; y = " << y << '\n';
  return 0;
}
```

Разыменование возвращенного функцией указателя на локальную не **static** переменную этой функции эквивалентно обращению по ссылке на эту переменную, т. е. порождает неопределенное поведение, поскольку после выхода из функции такие переменные уже не существуют.

8.2. Статические массивы

В случае, когда требуется группа однотипных значений известного размера, удобно воспользоваться средством, называемым *массив*.

Массив [*array*] — набор однотипных переменных, к каждой из которых можно обратиться по уникальному *индексу* [*index*[82]]. Переменные, составляющие массив, называются его *элемента-*

[82] От лат. *index* — «указатель, знак», «указательный палец».

ми. Общее количество элементов массива называется его *размером* [*size*].

Простейшей формой организации массивов в языке C++ являются одномерные статические массивы.

Слово *одномерный* означает, что индекс представляет собой скаляр (а именно, неотрицательное целое). У такого массива единственное *измерение* [*dimension*], имеющее размер [*extent*], равный размеру массива.

Пусть размер массива равен $n > 0$. Начальный элемент имеет индекс 0, следующий — 1 и так далее до последнего элемента, имеющего индекс $(n - 1)$.

Слово *статический* означает, что память под массив распределяется компилятором («статически»). Размер статического массива должен быть известен на момент компиляции (константа времени компиляции) и не может быть изменен во время работы программы.

Статический массив может быть как глобальным, так и локальным (локальной переменной в функции). Локальный не **static** массив создается каждый раз при исполнении функции и уничтожается при выходе из функции, как и любая другая подобная переменная.

Продемонстрируем определение статического массива и обращение к его элементам с помощью оператора [] (квадратные скобки) на простом примере:

```
#include <iostream>
using namespace std;
int main() {
  float data[10];
  // Считать данные из cin.
  for (int i = 0; i < 10; ++i)
    cin >> data[i]; // i-й элемент data
  // Вывести считанные данные в cout
  // в обратном порядке.
  for (int i = 0; i < 10; ++i)
    cout << data[9 - i];
}
```

В примере выше размер задан конкретным числом. Однако использование в таких целях чисел чревато ошибками: изменив размер массива в его определении, можно забыть изменить его в других местах. Если выполнять автоматическую замену числа в тексте, то каждый случай замены надо проверять, а для больших программ это неудобно. Поэтому лучше размеры статических массивов определять в виде именованных констант (времени компиляции) и затем везде использовать не конкретные числа, а их названия.

```
#include <iostream>
using namespace std;
int const DATA_SIZE = 10;
int main() {
  float data[DATA_SIZE];
  // Считать данные из cin.
  for (int i = 0; i < DATA_SIZE; ++i)
    cin >> data[i]; // i-й элемент data
  // Вывести считанные данные в cout
  // в обратном порядке.
  for (int i = 0; i < DATA_SIZE; ++i)
    cout << data[(DATA_SIZE - 1) - i];
}
```

Теперь, чтобы поменять размер массива data, достаточно сделать это в одном месте — в определении константы DATA_SIZE.

Статические массивы отражены в системе типов C++. Определение вида:

```
Type array[N];
```

задает переменную array типа Type[N]. Впрочем, синтаксис C++ **не** позволяет то же самое записать как:

```
Type[N] array; // в C++ это — синтаксическая ошибка
```

Можно определить несколько массивов разных размеров и других переменных «рядом»:

```
int n, *p, m, a[10], xy[2], &r = m;
```

Данная строка эквивалентна следующей группе определений:

```
int n;          // int
int * p;        // int*
int m;          // int
int a[10];      // int[10]
int xy[2];      // int[2]
int & r = m;    // int&
```

Выше мы не инициализировали массив data. Инициализатор массива выглядит следующим образом:

```
Type array[N] { элементы };
```

Значения элементов перечисляются через запятую в порядке индексов (что совпадает с порядком следования элементов в памяти).

В C++ необязательно указывать все элементы. Если элементов меньше, чем размер массива, то «хвост» массива инициализируется по умолчанию, т. е. заполняется нулями в случае встроенных типов. Наиболее распространенный вариант — инициализация всего массива нулями, когда список элементов инициализатора вовсе пуст (в этом случае «хвост» — весь массив):

```
float data[DATA_SIZE] {}; // заполнить нулями
```

На самом деле в современном C++ данная форма инициализатора применима к переменным любых типов:

```
int a = 0, b {0}, c {}; /* все три переменные
    инициализированы нулем. */
```

Пустой инициализатор {} означает «инициализировать значением по умолчанию», что в случае числовых типов означает 0 (**bool** — **false**), указателей — **nullptr**, массивов — инициализацию каждого элемента по умолчанию.

Если есть непустой инициализатор, то размер массива можно не указывать: он будет определен как равный числу элементов в инициализаторе:

```
uint32_t fib[] { 1, 1, 2, 3, 5, 8, 13 };
// размер fib равен 7
```

Использование такой конструкции поднимает вопрос о получении размера статического массива. Обычной ошибкой новичков является попытка использовать с этой целью оператор **sizeof**. Однако данный оператор возвращает размер своего аргумента в байтах, поэтому, например, если есть определение fib из примера выше, то **sizeof**(fib) будет равно 7***sizeof**(uint32_t), т. е. 28 на большинстве систем. Так как такой «размер» обычно больше истинного размера, то его использование вместо истинного размера влечет неопределенное поведение, поскольку C++ не гарантирует проверку выхода индекса за пределы допустимого диапазона.

Размер статического массива можно получить через отношение размера в байтах самого массива целиком к размеру в байтах одного его элемента:

```
sizeof(fib) / sizeof(fib[0])    /* индекс 0
    у статического массива всегда допустим. */
```

Но C++17 предлагает более удобный способ — стандартную функцию std::size (определена в заголовочном файле **iterator**):

```
#include <cstdint>
#include <iostream>
#include <iterator>
using namespace std;
int main() {
  uint32_t fib[] { 1, 1, 2, 3, 5, 8, 13 };
  for (size_t i = 0; i < size(fib); ++i)
    cout << i << ": " << fib[i] << '\n';
}
```

Массив символов можно инициализировать строковой константой:

```
char msg[80] = "(no message)";
```

Остаток массива также заполняется нулями, поэтому массив символов можно инициализировать нулями еще так:

```
char buf[1024] = "";
```

В случае, если строковая константа-инициализатор содержит больше символов, чем заявленный размер массива, компилятор должен сообщить об ошибке.

Сами строковые константы имеют тип массива **char** соответствующего размера. Например, "data" имеет тип **char**[5]. Размер его на единицу больше «видимого» числа символов, поскольку к такой строке всегда добавляется завершающий символ с кодом нуль, играющий роль признака конца строки. Поэтому, если не указать размер массива, то он будет равен размеру инициализатора:

```
char msg[] = "(no message)";
// msg имеет размер 13
// msg[12] равен '\0'
```

К строковой константе можно обращаться как к массиву:

```
for (int i = 0; i < 4; ++i)
  cout << "data"[i] << '\n';
```

Попытка изменить элемент строковой константы порождает неопределенное поведение:

```
"data"[0] = 'r'; // UB или ошибка компиляции
```

Попытка «одновременного» двойного изменения элемента массива также влечет неопределенное поведение:

```
int a = 0, arr[12] {};
arr[a = 10] = a; // UB
arr[0] = 0;
arr[arr[0]] = 1; // UB
```

8.3. Массивы и указатели

Через C и BCPL C++ унаследовал из языков низкого уровня тесную связь между массивами и указателями. На уровне машины массив есть просто блок памяти, а доступ к элементу по индексу осуществляется путем вычисления адреса элемента

как суммы (*база* + *смещение*), где *смещение* [*offset*] есть индекс, умноженный на размер элемента массива, а *база* [*base*] — адрес массива (его начального элемента) в памяти. Через оператор [] это записывается как base[index] (подразумевая сразу разыменование полученного указателя). Размер элемента является константой времени компиляции и выводится из типа указателя (или массива) base.

Итак, указатель может быть установлен на любой элемент массива, а сумма указателя и n (целого числа со знаком) дает указатель на элемент, сдвинутый на n позиций относительно исходного (вперед по индексам, если $n > 0$, назад, если $n < 0$).

Например:

```
int a[10] {};   // заполним нулями
int * p = &a[2];
*p = 10;        // a[2] = 10
*(p + 4) = 20;  // a[6] = 20
p += 5;         // p = &a[7]
*p = 30;        // a[7] = 30
```

Аналогично из указателя можно вычитать целое число, сдвигая его вперед, если число отрицательное, и назад, если оно положительное.

Если p — указатель, то p[i] есть то же самое, что и *(p+i). Более того, p[i] и i[p] интерпретируются компилятором одинаково. Это касается и встроенных массивов, так что 1["abc"] равно 'b'.

Складывать указатели друг с другом нельзя. Зато их можно вычитать друг из друга. Разность указателей есть целое число со знаком, соответствующее количеству элементов между ними. Знак определяется направлением отсчета элементов. Через разность можно ввести линейный порядок на множестве указателей на элементы одного массива: их можно сравнивать не только на равенство и неравенство, но и на «меньше», «больше» и т. д. Указатель a меньше указателя b, если их разность меньше нуля.

Вычисление разности указателей, не указывающих на элементы одного массива, влечет неопределенное поведение. Вычисление суммы указателя и смещения, приводящее к выходу указателя за пределы массива, влечет неопределенное поведение кроме той ситуации, когда полученный указатель указывает на мнимый элемент сразу за последним элементом массива, т. е. указатель на начальный элемент плюс размер массива (разыменовывать его, естественно, все равно нельзя). Такой указатель называют (*указатель на*) *конец массива.*

Разность указателей имеет целочисленный тип со знаком, имеющий ту же разрядность, что и стандартный тип size_t, используемый для представления размеров структур данных. В **cstddef** для него определен стандартный синоним ptrdiff_t[83].

Тот факт, что size_t не имеет знака, а ptrdiff_t имеет знак, формально влечет потенциальную возможность существования массива столь большого, что разность между указателями на его элементы может оказаться непредставима. На практике существующие системы не позволяют создать столь большой массив (больше половины адресного пространства).

Помимо вышеописанной «арифметики указателей» язык C++ унаследовал от C неявное приведение массива к указателю на него и указателю на начальный элемент массива, а также запрет на прямое копирование массивов через присваивание (влекущее невозможность передачи в функцию или возврата из функции массива по значению).

Итак, пусть N — константа времени компиляции и дано определение массива:

```
T a[N];
```

Тогда:

- T — тип элемента;
- T& — ссылка на элемент — тип выражения обращения по индексу, например, a[0]!;

[83] От англ. *pointer difference type* — «тип разности указателей».

- T* — указатель на элемент, например, &a[0], однако a и &a неявно приводятся к этому же типу со значением &a[0]!;

- T[N] — формальный тип переменной a;

- T(*)[N] — тип указателя на a, т. е. &a;

- T(&)[N] — тип ссылки на массив a.

Запись вида T(*)[N] или T(&)[N] может быть применена для определения переменной, имя которой надо поставить в скобках:

```
T arr[N];
T (&ref_to_arr)[N] = arr;   // ссылка на массив
T & err[N]; // ошибка: массив ссылок невозможен
T (*ptr_to_arr)[N] = &arr; // указатель на массив
T * arr_of_ptr[N];   // массив указателей на T
```

Возможно объявление синонима типа массива:

```
using I16 = int[16];
I16 a1, a2, a3; /* то же самое, что
  int a1[16], a2[16], a3[16]; */
```

Помимо функции std::size, возвращающей размер статического массива, в заголовке **iterator** определены еще две функции:

- std::begin возвращает указатель на начальный элемент операнда-массива (начало массива);

- std::end возвращает указатель на мнимый элемент, стоящий за последним элементом массива (конец массива).

Таким образом, (end(a) − begin(a)) равно ptrdiff_t(size(a)).

Массив можно передать в функцию по ссылке. В этом случае размер зафиксирован принимающей функцией как константа времени компиляции:

```
// Вычислить яркость цвета, заданного в модели RGB:
// rgb[i] ∈ [0.f, 1.f] — интенсивность i-го канала.
float luma(float (&rgb)[3]) {
  return .299f*rgb[0] + .587f*rgb[1] + .114f*rgb[2];
  // Может ли функция вернуть число > 1.f?
}
```

Внутри такой функции размер массива известен компилятору и, например, size(rgb) будет константой времени компиляции, равной 3. Операндом luma может быть только массив **float**[3].

Впрочем, C++ позволяет и иную запись:

```
float luma(float rgb[3]) {
  return .299f*rgb[0] + .587f*rgb[1] + .114f*rgb[2];
}
```

Эта обманчивая запись эквивалентна двум другим вариантам:

```
float luma(float rgb[]) { ... }
float luma(float * rgb) { ... }
```

Для компилятора эти три варианта объявления параметра rgb одинаковы. В такой записи размер массива (да и само указание на то, что это массив) является информацией для человека: во всех трех случаях rgb будет иметь тип **float***! Функции begin, end и size уже нельзя будет вызвать для rgb, так как они не определены для указателей (поскольку не существует общего способа только по самому указателю узнать размер массива, на элемент которого указывает этот указатель).

Передача массива по указателю возможна, но опасна. Если мы используем одно из трех последних определений функции luma, то следующий код скомпилируется, возможно, даже без предупреждений:

```
float xy[2] { 0.5f, 1.0f };
cout << luma(xy); // UB: xy[2] не определен!
```

Итак, при определении функции, которая должна принимать массив, у нас есть три основных опции:

1. Передать массив заранее фиксированного в функции размера по ссылке.

2. Передать указатель на начальный элемент массива и размер массива (отдельным параметром).

3. Передать указатели на начало и конец массива (их разность дает размер массива).

Для функции luma выше следовало выбрать первую опцию, поскольку цвет задан тремя каналами (что есть следствие человеческой физиологии). Первой опции соответствует первый вариант определения luma.

Иную опцию следует выбрать, если мы хотим, чтобы функция могла обрабатывать массивы разных размеров (или даже части массивов). Например, функцию, вычисляющую сумму массива, можно определить так (опция 3):

```
int sum(int const * from, int const * to) {
  int s = 0;
  while (from < to)
    s += *from++;
  return s;
}
```

(Запись *from++ означает «передвинуть указатель from на следующий элемент, а разыменовать старое значение указателя».)

Теперь, например, можно вычислить сумму массива так:

```
int a[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
cout << sum(std::begin(a), std::end(a)); // 45
// Обработать только часть массива:
cout << sum(&a[2], &a[5]); // 12
```

Данный способ широко используется в стандартной библиотеке C++ (точнее ее части, созданной на основе некогда отдельной библиотеки под названием STL).

Опция 2 может быть реализована так:

```
int sum(int const * arr, size_t sz) {
  int s = 0;
  for (size_t i = 0; i < sz; ++i)
    s += arr[i];
  return s; }
```

```
...
  int a[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  cout << sum(std::begin(a), std::size(a)); // 45
```

Вместо возвращения массива из функции как единого значения можно принять результирующий массив в качестве параметра (или двух параметров) и заполнить его внутри функции. Выше все варианты luma могли это сделать, изменив значения элементов rgb. Следовало определить luma как принимающую массив констант:

```
// Вычислить яркость.
float luma(float const (&rgb)[3]) {
  return .299f*rgb[0] + .587f*rgb[1] + .114f*rgb[2];
}

// Конвертировать цвет в ч/б на месте.
void to_luma(float (&rgb)[3]) {
  rgb[0] = rgb[1] = rgb[2] = luma(rgb);
}
```

«Возвращение» массива, заполненного членами арифметической прогрессии:

```
// Возвращает конец заполненного диапазона.
double * linspace(
    double a0, double delta,
    size_t n, double a[]) {
  for (size_t i = 0; i < n; ++i)
    a[i] = a0 + delta * i;
  return a + n;
}
```

Код, вызывающий данную функцию, должен передать ей указатель на массив, в котором достаточно места для размещения элементов. Сама функция никак не может узнать, что массив кончился.

8.4. О линейном поиске

Линейным поиском называют поиск какого-либо значения в наборе перебором всех элементов набора.

Например, линейный поиск значения в массиве легко записать с помощью **for**:

```
// Возвращает позицию найденного элемента.
// Если значение не найдено, возвращает nullptr.
int * find(int values[], size_t n, int value) {
  for (size_t i = 0; i < n; ++i)
    if (values[i] == value)
      return values + i;
  return nullptr;
}
```

Похожий код поиска на диапазоне, заданном парой указателей, удобно записать с помощью **while**:

```
// Возвращает позицию найденного элемента.
// Если значение не найдено, возвращает to.
int * find(int * from, int * to, int value) {
  while (from != to && *from != value)
    ++from;
  return from;
}
```

Данная конструкция является типичной для операций линейного поиска и базирующихся на них проверок кванторов существования и всеобщности. Например, вычисление истинности высказывания «существует элемент диапазона, равный value» переводится в следующий код:

```
bool exists(int * from, int * to, int value) {
  return find(from, to, value) != to;
}
```

У Как будет выглядеть аналогичная функция exists для массива (используя первый вариант функции find)?

Высказывание «не существует элемент диапазона, равный value» есть отрицание предыдущего:

```
bool not_exists(int * from, int * to, int value) {
  return find(from, to, value) == to;
  // return !exists(from, to, value);
}
```

Квантор всеобщности выражается через квантор существования с помощью двух отрицаний:

$$\forall x\ P(x) \equiv \neg\exists x\ \neg P(x).$$

Поэтому высказывание «все элементы диапазона равны value» эквивалентно высказыванию «не существует элемента диапазона, не равного value». Оба варианта можно перевести на C++ независимо:

Пример 8.1. Всеобщность через поиск

```
int * find_not_eq(int * from, int * to, int value) {
  while (from != to && *from == value) // != → ==
    ++from;
  return from;
}

// Все элементы [from, to) равны value?
bool all_eq(int * from, int * to, int value) {
  return find_not_eq(from, to, value) == to;
}
```

Если внести определение функции find_not_eq в определение функции all_eq, то получим:

Пример 8.2. Всеобщность непосредственно

```
bool all_eq(int * from, int * to, int value) {
  for (; from != to; ++from)
    if (*from != value)
      return false;
  return true;
}
```

У Записать аналогичные примерам 8.1 и 8.2 определения для высказывания «не все элементы диапазона равны value».

Вместо проверки на равенство заданному значению можно использовать любой предикат.

8.5. Цикл for для диапазона

Стандарт C++11 определил еще один вид цикла, удобный при работе с массивами и другими структурами данных, которые можно использовать как последовательности однотипных значений.

Цикл по диапазону [*range-for*] — цикл «для всех», т. е. перебирающий все элементы *диапазона*.

Диапазон [*range*] — объект, для которого определены функции begin и end, возвращающие значения, которые можно *использовать как* указатели на элементы последовательности.

В C++ цикл по диапазону имеет следующий синтаксис:

```
for (переменная : диапазон)
  тело-цикла
```

где диапазон есть выражение, значение которого может быть использовано как диапазон.

Данная запись по принципу действия эквивалентна следующему коду:

Пример 8.3. Симуляция range-for

```
{
  using std::begin;
  using std::end;
  auto && __range = диапазон;
  auto __begin = begin(__range);
  auto __end = end(__range);
  for (; __begin != __end; ++__begin) {
    переменная = *__begin;
    тело-цикла
  }
}
```

Смысл конструкции вида **auto**&& заключается в том, что __range будет значением (копией), если диапазон вычисляется

как временное значение, и ссылкой, если диапазон вычисляется как ссылка.

Массив является частным случаем диапазона:

```
int a[] { 1, 2, 3, 4, 5 };
// Возвести в квадрат все элементы a.
for (int & x : a)
  x *= x;
// Вывести все элементы a.
for (int x : a)
  cout << x << ' ';
```

Так как строковая константа является массивом, ее можно использовать в качестве диапазона:

```
// Вывести каждый символ и его код.
for (char ch : "Hello!")
  cout << ch << " : " << int(ch) << '\n';
```

Наконец, в качестве диапазона можно использовать инициализатор массива[84], что бывает удобно, когда требуется перебрать в цикле непосредственно заданную последовательность значений:

Пример 8.4. Без continue v.2

```
// Четыре угла.
for (int dx : { -1, +1 })
  for (int dy : { -1, +1 })
    cout << "(" << dx << ", " << dy << ")\n";
```

У Как изменить последнюю строчку в примере 8.4, чтобы получить четыре направления $(1,0)$, $(0,1)$, $(-1,0)$ и $(0,-1)$ (в произвольном порядке)?

Задания для самопроверки

У Ответьте на следующие вопросы.

[84] Чтобы код скомпилировался, может потребоваться подключение заголовка **initializer_list**.

- Какими будут размеры массивов в следующих двух определениях?

```
char data[] { 0, 1, 2, 3, 4 };
char str[] = "01234";
```

- Какое из двух определений допустимо (при каком-либо подходящем инициализаторе *init*)?

```
int &* ptr2ref = init;
int *& ref2ptr = init;
```

  Чем здесь может быть инициализатор?

- Чем отличаются друг от друга два следующих определения?

```
char * arr1[100];   // вариант 1
char (*arr2)[100]; // вариант 2
```

- Допустимо ли определение вида:

```
int & arr[10];
```

  Почему?

- Как написать цикл, перебирающий все символы заданной строковой константы?

- В чем ошибка автора следующей функции?

```
float sum(float const x[]) {
  float s = 0;
  for (size_t i = 0; i < sizeof(x); ++i)
    s += x[i];
  return s;
}
```

  Как ее можно исправить?

У Напишите функцию, вычисляющую истинность высказывания «не все элементы массива положительны».

У Напишите функцию, вычисляющую истинность высказывания «среди элементов массива есть четные числа».

Глава 9

Управление памятью

9.1. Классификация переменных

Классификация переменных (значений в памяти) возможна по целому ряду признаков. Мы уже видели, что с точки зрения возможности изменения переменная может быть:

- изменяемой (по умолчанию);
- константной («только для чтения»), если ее тип имеет модификатор **const**.

Также ранее мы рассматривали разные режимы связывания переменных и функций:

- нет связывания (переменная определена внутри функции) — *локальные переменные* [*local variables*];
- внутреннее связывание (модификатор **static** или расположение в анонимном пространстве имен) — *переменные модуля*;
- внешнее связывание (прочие переменные) — *глобальные переменные* [*global variables*].

Локальные переменные доступны только из своей функции. Переменные модуля доступны только из своего модуля (единицы трансляции). Глобальные переменные потенциально доступны отовсюду.

Кроме связывания термины «глобальный» и «локальный» также применяются к *области видимости* [*scope*] имен. Глобальная область видимости содержит все остальные области видимости и доступна отовсюду.

Каждое пространство имен вводит свою область видимости, которая для размещенных в ней определений является локальной[85], поскольку используется по умолчанию, *затеняя* [*shadowing*] внешнюю область видимости — если объявить в ней имя, совпадающее с уже существующими внешними именами, то внешние имена становятся не видны.

Тело каждой функции также вводит свою область видимости. Далее, каждый блок вводит свою область видимости. Так как в процессе исполнения программы локальные переменные то «появляются» в текущей области видимости исполнителя программы, то «исчезают» (с выходом из соответствующего блока или функции), то важным является вопрос о сохранении или потере значения между их повторными появлениями в текущей области видимости исполнителя (будет ли это одна и та же переменная или, по сути, разные под одним названием). **Класс хранения** [*storage class*] — способ выделения памяти под переменную. Класс хранения определяет время жизни переменной.

В C++ есть два основных класса хранения локальных переменных:

- *автоматический* (по умолчанию) — переменная создается, когда встречается ее определение, и уничтожается, когда ее имя покидает область видимости;
- *статический* (с модификатором **static**) — переменная создается, когда ее определение встречается **первый раз**

[85] От лат. *locus* — «место».

при исполнении кода, и уничтожается при завершении программы. Глобальная переменная всегда имеет статический класс хранения.

Данное деление основано на способе выделения памяти компилятором. Под автоматические переменные память выделяется временно по мере необходимости («автоматически»), обычно для этого используются регистры процессора и *стек вызовов*. Под статические переменные память распределяется компилятором заранее («статически») и выделяется в момент запуска программы (загрузки исполняемого файла). Адрес статической переменной является константой времени компиляции.

В отличие от автоматических, статические переменные всегда инициализируются (нулями). Следующая функция использует статическую локальную переменную для подсчета числа вызовов себя:

```
void greet() {
  static unsigned calls;
  cout << "Greets";
  if (++calls == 100)
    cout << ", 100th time";
  cout << "!\n";
}
```

Статические локальные переменные имеют определенную популярность (особенно в C), так как иногда позволяют (или «позволяют») избежать сложностей с управлением памятью.

Например, функцию, конвертирующую число в его двоичную запись, можно написать так:

```
char const * bin(uint32_t num) {
  static int const bits = 32;
  static char d[bits+1];
  for (int i = 0; i < bits; ++i)
    d[(bits-1)-i] = ((num >> i) & 1? '1': '0');
  return d;
}
```

Здесь bin возвращает всегда одно и то же значение — константу времени компиляции, а именно адрес bits. По сути, это скрытая глобальная переменная, прямой доступ к которой есть только у функции bin. К сожалению, из-за этого функция bin не является чистой, а значит, возможны неожиданности. Например, если bin будет вызвана одновременно из двух *потоков исполнения*, то содержимое bits будет испорчено.

Для решения последней проблемы в C++11 введен еще один класс хранения: *переменная потока* [*thread storage duration*], ключевое слово **thread_local**. Этот класс хранения подразумевает статическое распределение памяти.

```
char const * bin(uint32_t num) {
  static int const bits = 32;
  thread_local char d[bits+1];
  for (int i = 0; i < bits; ++i)
    d[(bits-1)-i] = ((num >> i) & 1? '1': '0');
  return d;
}
```

Теперь bin можно вызывать параллельно, и каждый поток исполнения получит свой указатель. Впрочем, данный указатель все равно не следует делать доступным другим потокам исполнения или хранить (так как любой следующий вызов bin заменит содержимое массива, на который он указывает).

Итак, рекомендуется избегать возврата ссылок или указателей на изменяемые статические локальные переменные.

9.2. Стек вызовов

Стек вызовов [*call stack*] представляет собой часть механизма вызова функций и является областью памяти, выделяемой при запуске приложения и освобождаемой по завершении работы, которая служит для хранения адресов возврата (из вызванной функции в вызвавший код), аргументов, результатов, локальных переменных и вспомогательных данных, размещаемых компилятором.

Размер стека вызовов обычно нельзя менять во время работы, поэтому есть опасность *переполнения стека вызовов* [*call stack overflow*] в случае слишком длинной цепочки вызовов функций или при использовании переменных автоматического класса хранения большого размера. С точки зрения C++ данная ошибка влечет неопределенное поведение, в операционных системах с защитой памяти переполнение стека вызывает принудительное завершение работы приложения.

Заполненная часть стека вызовов состоит из *кадров* [*call stack frames*] — кусочков, хранящих данные одного вызова. Самый свежий кадр находится на *вершине стека* [*call stack top*] и отвечает вызову функции, которая исполняется в данный момент.

Таким образом, при вызове функции над текущей вершиной стека (в свободном пространстве стека вызовов) создается новый кадр, который становится вершиной стека после перехода процессора к исполнению кода вызванной функции. При возврате из функции кадр стека на вершине уничтожается и снова становится свободным пространством, а вершиной становится кадр под ним, отвечающий функции, к продолжению исполнения кода которой возвращается процессор. Поэтому можно представлять стек вызовов как «стопку» кадров, где каждый кадр является «снимком» состояния данных какой-то из функций.

Типичный алгоритм вызова функции можно описать так:

1. Записать над вершиной стека все вспомогательные данные, сохранение которых должен гарантировать вызов. Обычно это копии некоторых регистров процессора, т. е. часть состояния самого процессора, которую необходимо будет затем восстановить.

2. Сверху записать все аргументы по очереди.

3. При необходимости зарезервировать место для результата.

4. Над аргументами записать *адрес возврата* [*return address*], т. е. адрес инструкции машинного кода *вызывающей функции* [*caller*], на которую должен вернуться процессор из *вызываемой функции* [*callee*]. Новый кадр стека вызовов считается созданным.

5. Осуществить переход на первую инструкцию машинного кода вызываемой функции, приступив, таким образом, к ее исполнению.

6. Вызванная функция обращается к аргументам, используя фиксированные смещения (для этого компилятор должен знать размеры аргументов в байтах) от указателя на первый свободный байт стека (как правило, этот указатель хранится в специальном регистре процессора, называемом *указателем стека* [*stack pointer*]).

7. Вызванная функция может произвольным образом распоряжаться свободной памятью стека (обычно путем размещения там локальных переменных), но при очередном вызове, естественно, создает новый кадр над уже используемым пространством.

8. При возврате вызванная функция формирует значение результата и выполняет все необходимые действия по деинициализации локальных переменных. Иногда никаких действий не требуется — занятая локальными переменными память просто объявляется свободной путем изменения указателя стека. В таком случае значения остаются в памяти и могут быть доступны другим функциям. В этой простоте заключается дешевизна управления автоматической памятью.

9. Окончательный возврат и уничтожение кадра стека происходит при загрузке сохраненного адреса возврата и переходе на него. Далее выполняется код после точки вызова, который загружает сохраненные на первом шаге данные и работает с результатом вызова.

Конкретика алгоритма вызова зависит от архитектуры процессора и от операционной системы, а в языках высокого уровня реализуется транслятором.

Соблюдение общих правил вызова позволяет функциям, написанным на разных языках программирования, вызывать друг друга. Обычно эти правила (*соглашение вызова*) базируются на семантике языка C и входят в стандарт *двоичного интерфейса приложений* данной платформы [*application binary interface, ABI*].

Прямое изменение данных в стеке вызовов может влечь неопределенное поведение. Например, запись данных в автоматически размещаемый массив по индексу вне допустимого диапазона может изменить некие данные в стеке вызовов, которыми могут оказаться не только значения переменных, но и, например, адрес возврата, из-за чего возврат из функции может произойти неизвестно куда. Обычно это влечет «падение» программы, но может использоваться и в целях «взлома» программы перенаправлением возврата из функции на заданный взломщиком адрес в памяти.

9.3. Динамическая память

Динамическая память — механизм, позволяющий непосредственно в ходе исполнения программы распределять и перераспределять участки памяти. В частности, выделять блоки размера, определяемого во время исполнения программы.

В C++ имеется две реализации динамической памяти: подсистема C, реализованная в виде части стандартной библиотеки, и встроенная в язык подсистема C++, реализованная в виде операторов **new** и **delete**. Данные подсистемы не обязаны быть совместимы: попытка освободить или перераспределить средствами одной подсистемы блок памяти, выделенной средствами другой подсистемы, влечет неопределенное поведение.

Рассмотрим подсистему C++.

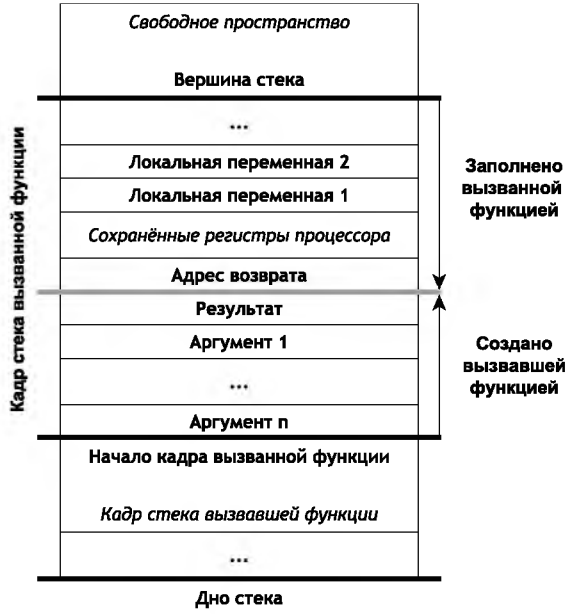


Рис. 9.1. Кадр стека вызовов

Оператор **new** позволяет создать в динамической памяти новую переменную. По сути он выполняет два действия: выделение свободного блока памяти подходящего размера и, если требуется, инициализацию переменной (запись некоторого начального представления). Результат выражения **new** есть указатель на созданную переменную.

```
T * pvar = new T; /* инициализация по умолчанию
не инициализирует встроенные типы. */
int * ivar = new int { 10 }; // инициализатор: 10
```

Созданная таким образом переменная будет существовать, пока ее не удалят с помощью оператора **delete**, либо пока исполнение программы не будет завершено:

```
delete pvar;
delete ivar;
```

Рис. 9.2. Общая структура памяти процесса

Попытка применить **delete** к неинициализированному указателю и попытка дважды (и более раз) удалить одну и ту же переменную влекут неопределенное поведение.

Оператор **delete** можно применить к нулевому указателю, данное действие не влечет никаких эффектов.

```
int * nothing = nullptr;
delete nothing; // ничего не делает
```

Есть вторая форма операторов **new** и **delete**, предназначенная для создания и освобождения массивов. Оператор **new[]**:

```
// Инициализация по умолчанию:
T * parr = new T[size];
// Динамический массив размера 4, заполненный
// значениями 1, 2, 3, 0.
int * iarr = new T[4] { 1, 2, 3 };
```

Так же, как и в случае статического массива, можно использовать пустой инициализатор {} для заполнения новосозданного массива нулями.

Массив, созданный в динамической памяти, называют *динамическим*[86].

Для освобождения динамического массива следует использовать оператор **delete[]**:

```
delete[] parr;
delete[] iarr;
```

Попытка удалить массив «простым» оператором **delete** и, наоборот, попытка удалить оператором **delete[]** переменную, созданную «обычным» оператором **new**, влечет неопределенное поведение.

Как правило, при нормальной работе программы каждый выделенный блок должен быть впоследствии освобожден, в противном случае возникает *утечка памяти.*

Утечка памяти [*memory leak*] — уменьшение объема памяти, доступного программе, вызванное ошибками в самой программе. Как правило, происходит при выделении блоков динамической памяти, адреса которых затем по какой-то причине теряются, из-за чего программа не может ни использовать их, ни вернуть системе эти блоки памяти.

В случае невозможности выделить кусок памяти нужного размера **new** бросает исключение std::bad_alloc. Таким образом, если исполнение программы продолжилось, то мы знаем, что память была выделена успешно, и возвращенным указателем можно пользоваться.

Чтобы отключить бросание исключения, надо подключить заголовок **new** и записать: **new**(std::nothrow) T. В случае неудачи операция вернет нулевой указатель.

---

[86] Часто под «динамическим» понимается массив, размер которого можно изменять во время исполнения программы. К сожалению, нет стандартных средств, позволяющих изменить размеры блока памяти (и/или массива), выделенного оператором **new**.

Пример 9.1. Задача чтения массива: new и delete

```
#include <new> // std::nothrow
// Прочитать массив заранее неизвестного размера.
// Читать элементы до первой ошибки.
// Возвращенный массив удалить через delete[]!
double* read_nums(size_t & size) {
  size_t capacity = 15;
  auto nums = new double[capacity];
  size = 0;
  while (cin >> nums[size]) {
    if (++size < capacity) continue;
    capacity = 2*capacity + 1;
    auto new_nums = new(nothrow) double[capacity];
    if (!new_nums) break;
    for (size_t i = 0; i < size; ++i)
      new_nums[i] = nums[i];
    delete[] nums;
    nums = new_nums;
  }
  return nums;
}
```

Данный пример носит чисто учебный характер: при использовании C++ следует либо использовать готовые компоненты стандартной библиотеки, либо реализовать свой расширяемый массив, либо использовать связанный список массивов.

9.4. Тип string

Как мы уже видели, строковые константы в C++ имеют тип «массив символов», из-за чего работать с ними может быть неудобно.

Стандартная библиотека C++ предлагает тип std::string, определенный в заголовочном файле **string**, который представляет собой удобный интерфейс поверх динамического массива символов. Он берет на себя операции выделения, освобождения и копирования памяти, поэтому пользователь может оперировать объектами string как обычными значениями вроде целых

чисел. В частности, можно возвращать из функции объекты string по значению:

```
string bin(uint32_t num) {
  static int const bits = 32;
  char d[bits];
  for (int i = 0; i < bits; ++i)
    d[(bits-1)-i] = ((num >> i) & 1? '1': '0');
  return string(d, bits);
}
```

Эту версию функции bin «в первом приближении[87]» можно считать чистой функцией.

В данной книге string не рассматривается подробно. Опишем некоторые базовые операции.

Объекты string можно создавать из строковых констант, массивов символов и отдельных символов. Все следующие инициализаторы дадут один и тот же результат (строку «abc»):

```
string s1 = "abc";
string s2 { "abc" };
string s3("abc");
string s4 { 'a', 'b', 'c' };

char data[] = "abc";
string s5(data, 3); // 3 — сколько символов
```

В отличие от переменных встроенных типов C++ переменные string при отсутствии инициализатора автоматически инициализируются пустой строкой.

Также можно создать строку, состоящую из повторяющегося символа:

```
string ast(5, '*');
cout << ast; // *****
```

[87] Динамическая память является глобальным разделяемым ресурсом. Потенциально возможна ошибка выделения памяти при попытке создать очередной объект string.

Строки можно сравнивать друг с другом и со строковыми константами. На строках определен линейный лексикографический («словарный») порядок:

```
cout << (string(3, 'a') < "b"); // 1: aaa < b
```

К строкам можно обращаться как к массивам символов с помощью оператора []. Размер строки можно узнать с помощью функции size. Изменить размер можно с помощью функции resize (символы добавляются и удаляются с конца строки):

```
string a("hello!", 4);
cout << a.size(); // 4
cout << size(a);  // 4
a[0] = 'w';
a.resize(5);
a[4] = '!';
cout << a; // well!
```

Для объектов string доступен цикл **for** по диапазону. Так как копирование строк — сравнительно дорогая операция, рекомендуется передавать строки в функции по **const**-ссылке, а не по значению:

```
// Посчитать, сколько раз встречается символ
// с заданным кодом в строке.
size_t count(string const & s, char ch) {
  size_t result = 0;
  for (char x : s)
    result += x == ch;
  return result;
}
```

Значения ряда типов (в частности, чи́сла) можно преобразовывать в строковые представления с помощью стандартной функции to_string:

```
cout << (to_string(0xFE) == "254"); // 1
```

Распознавание строковых представлений целых чисел можно делать с помощью стандартных функций stoi (возвращает **int**), stol (**long**), stoll (**long long**), stoul (**unsigned long**), stoull (**unsigned long long**), имеющих одинаковый набор параметров:

```
int stoi(string const & s,
    size_t * pos = nullptr,
    int radix = 10);
```

По адресу pos (если это не нулевой указатель) сохраняется индекс первого символа за считанным числом. Значение radix задает основание системы счисления (принимаются основания от 2 до 36). Если в качестве значения radix передать 0, то основание будет распознаваться по префиксу (0 — основание 8, 0x и 0X — 16, другое — 10).

В случае, если запись числа начинается на знак «минус», функции stoul и stoull возвращают результат применения операции отрицания к соответствующему беззнаковому числу. Например, вызов stoul("−10000") возвращает 4 294 957 296.

Для распознавания строковых представлений дробных чисел имеются стандартные функции stof (возвращает **float**), stod (**double**), stold (**long double**), имеющие одинаковый набор параметров:

```
int stof(string const & s,
    size_t * pos = nullptr);
```

По адресу pos (если это не нулевой указатель) сохраняется индекс первого символа за считанным числом. Числа распознаются в соответствии с правилами записи констант C, кроме того, принимаются строки INF или INFINITY (без учета регистра) как значение «бесконечность» и NAN как значение «нечисло».

Данные функции бросают исключение в случае ошибки (см. с. 486). На практике может быть удобнее использовать их прообразы из стандартной библиотеки C (с. 293).

Для стандартных строк определена операция *конкатенации* (приписывания или «сложения» строк):

```
int x = 12, y = 23;
cout << (to_string(x) + ", " + to_string(y));
// 12, 23
string a = "a";
a += a;
```

```
a += a;
cout << a.size(); // 4: aaaa
```

Цикл чтения строк в стандартной форме имеет две версии: чтение «слов» (последовательностей непробельных символов) и строк[88] (последовательностей символов, разделенных переводами строк).

Пример 9.2. Чтение слов

```
for (string word; cin >> word;)
  cout << word << "␣:␣" << word.size() << '\n';
```

У Запустите пример 9.2 и поэкспериментируйте с ним.

Чтение строки до заданного символа обеспечивается функцией getline, принимающей три параметра: поток ввода, строку (по ссылке), в которую записать прочитанные символы, и символ — разделитель строк (можно не передавать: по умолчанию это символ перевода строки). Функция возвращает поток ввода.

Пример 9.3. Чтение строк

```
for (string line; getline(cin, line);)
  cout << line << '\n' << line.size() << '\n';
```

У Запустите пример 9.3 и поэкспериментируйте с ним.

9.5. Динамическая память: средства C

Бестúповый указатель [*untyped pointer*] — указатель, для которого не определен тип значения, на которое он указывает.

Все указатели, с которыми мы имели дело ранее, были *типизированными* [*typed*] — тип значения, на которое они указывают, был «зашит» в типе самого указателя. Например, **int*** — указатель на **int**.

[88] В английском языке в этом случае используется слово *line* — «строка текста».

На низком уровне иногда удобно оперировать просто адресами участков памяти без определения типа значения, которое должно быть там расположено. В этом случае вместо типа значения следует поставить **void**. Тип бестипового указателя записывается **void***, его множеством значений являются все указатели (любой указатель можно привести к **void***).

Первоначально в качестве бестипового в языке C использовался указатель на байт, т. е. **char***. И сейчас в C++ допустимо использовать **char*** для «просмотра» двоичного представления значения произвольного типа T как массива **char**[**sizeof**(T)].

Но **void*** является более абстрактным: все, что с ним можно делать, это сравнивать на равенство или неравенство с другими указателями и явно приводить к типам других указателей. При этом в ряде случаев можно «получить» неопределенное поведение:

```
double x = 1.0;
void * p = &x;              // можно
cout << *(double*)(p); // 1, можно
cout << *(int*)(p);        // ?, UB!
```

Подсистема динамической памяти языка C доступна через заголовочный файл **cstdlib** и включает пять функций:

- **void*** malloc(size_t bytes) — выделяет в динамической памяти блок размера bytes байт и возвращает его адрес. Если выделить блок запрошенного размера не удалось, то malloc возвращает **nullptr**.

- **void*** calloc(size_t num, size_t size) — выделяет в динамической памяти блок размера (num*size) байт, обнуляет его содержимое и возвращает его адрес. Также возвращает **nullptr** в случае неудачи.

- **void*** realloc(**void*** old, size_t new_bytes) — перераспределяет ранее выделенную память (изменяет размер выделенного блока). Если передать в качестве old нулевой указатель, то вызов эквивалентен malloc(new_bytes). Если передать в качестве

old указатель, полученный ранее от malloc, calloc или realloc, то данная функция пытается:

- изменить размер выделенного блока на new_bytes;
- если это невозможно, выделить новый блок памяти размера new_bytes, скопировать туда содержимое старого блока и освободить старый блок.

В случае неуспеха realloc возвращает нулевой указатель, оставляя старый блок неизменным.

- **void**∗ aligned_alloc(size_t alig, size_t bytes) (C11, C++17) — выделяет блок размера bytes байт по адресу, кратному alig (*выровненному по* alig), и возвращает его адрес. В случае неуспеха возвращает **nullptr**.

- **void** free(**void**∗) — освобождает блок памяти, выделенный одной из вышеперечисленных функций. В случае передачи нулевого указателя ничего не делает.

Поведение не определено, если:

- передать функции free или realloc указатель, который не был ранее получен вызовом функции malloc, calloc, realloc или aligned_alloc;
- разыменовать указатель, ранее переданный функции free;
- дважды передать функции free один и тот же указатель на ранее выделенный блок (двойное удаление);
- использовать оператор **delete** для освобождения блока памяти, выделенного вызовом функции malloc, calloc, realloc или aligned_alloc.

Не следует использовать realloc для освобождения памяти, так как если передать ей новый размер, равный нулю, то поведение определяется реализацией.

Не следует также использовать realloc для перераспределения блока, выделенного aligned_alloc, поскольку realloc не гарантирует сохранность выравнивания.

Типичные ошибки:

```
// Возможно «тихое» переполнение:
auto p = (int*)malloc(n*sizeof(int));
// Утечка памяти в случае неудачи:
buf = realloc(buf, new_size);
```

Следует писать:

```
// Неудача в случае переполнения:
auto p = (int*)calloc(n, sizeof(int));
// Проверить на неудачу:
if (auto new_buf = realloc(buf, new_size))
  buf = new_buf;
else ... // неудача, buf сохранен
```

К сожалению, нет стандартных функций, совмещающих в себе возможности сочетаний aligned_alloc, calloc и realloc.

У Попробуйте определить функцию crealloc:

```
void* crealloc(void* old,
        size_t new_elems, size_t elem_size);
```

проверяющую (new_elems*size) на переполнение и вызывающую realloc в случае успеха. К сожалению, используя стандартные средства, мы не можем узнать размер старого блока, чтобы обнулить добавленную часть.

Приведем пример использования crealloc:

```
// Прочитать массив заранее неизвестного размера.
// Читать элементы до первой ошибки.
// Возвращенный массив удалить через free!
double* read_nums(size_t & size) {
  size_t capacity = 15;
  auto nums =
    (double*)calloc(capacity, sizeof(double));
  size = 0;
  while (cin >> nums[size]) {
    if (++size < capacity) continue;
```

```
    capacity = 2*capacity + 1;
    if (auto new_nums = (double*)crealloc(
          nums, capacity, sizeof(double)))
      nums = new_nums;
    else
      break;
  }
  return nums;
}
```

Теперь нам не нужно вручную копировать старый массив, поскольку (c)realloc делает это за нас.

Необходимо отметить, что в отличие от подсистемы динамической памяти C подсистема динамической памяти C++ гарантирует минимальную необходимую инициализацию типов C++, не являющихся *примитивными*, т. е. определимыми средствами языка C. Поэтому, например, можно создать массив string с помощью **new**:

```
auto sa = new string[100];
sa[0] = "ok!";
...
delete[] sa;
```

Однако попытка выполнить аналогичные действия средствами подсистемы C повлечет неопределенное поведение:

```
auto sa = (string*)calloc(100, sizeof(string));
sa[0] = "ok!"; // UB!
...
free(sa); // ???
```

Задания для самопроверки

У Ответьте на следующие вопросы.

- Обязана ли статическая константа быть константой времени компиляции?

- Может ли **thread_local**-переменная иметь автоматический класс хранения?
- Может ли **thread_local**-переменная размещаться в стеке вызовов?
- Что называют «переполнением стека вызовов»? Что может повлечь переполнение стека вызовов?
- Возможна ли утечка памяти при вызове определенной ниже функции f?

```
void f(int buf_sz) {
  auto buf1 = new char[buf_sz];
  auto buf2 = new char[buf_sz];
  // ... (какой-то код, не содержащий ошибок)
  delete[] buf2;
  delete[] buf1;
}
```

- Где ошибка в следующем коде?

```
auto buf = new char[N];
// ...
buf = realloc(buf, 2*N);
// ...
```

У Попробуйте написать программу, выводящую содержимое занятой части стека вызовов.

У Запишите выражение, создающее объект string, содержащий строку из 80 знаков = и одного знака перевода строки.

У Используя функцию getline, напишите инструкцию, читающую из потока cin все содержимое до символа «!».

У Рассмотрим следующий пример кода:

```
#include <string>
#include <iostream>
using namespace std;
```

```
string author_name(
    string const & first_name,
    string const & second_name) {
  return second_name + ", " + first_name[0] + ".";
}

int main() {
  string name, surname;
  cout << "name = ";
  cin >> name;
  cout << "surname = ";
  cin >> surname;
  cout << "author_name = "
       << author_name(name, surname) << endl;
  return 0;
}
```

Замените функцию author_name на функцию initials, возвращающую для переданных ей «Имя» и «Фамилия» строку вида «И.Ф.».

У Напишите функцию copybuf, создающую в динамической памяти копию переданного ей массива и возвращающую адрес этой копии:

```
char * copybuf(char const buf[], size_t size);
```

У Напишите функцию page_alloc, выделяющую n страниц размера 4кб (адрес страницы выровнен по границе 4 096 байт):

```
void * page_alloc(size_t n);
```

Глава 10

Составные типы данных

10.1. Многомерные статические массивы

Стандартная схема организации многомерных массивов заключается в выражении их как массивов массивов (массивов...). Рассмотрим это на примере двумерного статического массива.

```
// T — некоторый тип (элемента массива).
T array[N][M]; // N и M — размерности
```

Данный массив расположен в памяти одним куском из NM элементов типа T. Так как это массив массивов, то обращение по первому индексу дает нам подмассив размера M:

```
using Subarray = T[M];
Subarray & ai = array[i];
// i в пределах от 0 до (N − 1)
```

Соответственно, подмассив автоматически приводится к указателю на T[M] и далее к указателю на T, установленному на (&array[0][0] + i*M).

Инициализатор двумерного массива может быть построен как последовательность инициализаторов подмассивов:

```
int a[2][3] { { 1, 2, 3 }, { 4, 5, 6 } };
```

Альтернативный вариант — «сплошной» инициализатор (элементы заполняются в порядке их расположения в памяти):

```
int a[2][3] { 1, 2, 3, 4, 5, 6 };
```

В данном примере оба инициализатора дают одинаковый результат.

Однако иногда результат может быть различным:

```
int a[2][3] {
  { 1, 2 },   // то же, что 1, 2, 0
  { 4 }     }; // то же, что 4, 0, 0
```

эквивалентно

```
int a[2][3] { 1, 2, 0, 4 };
// Остаток заполняется нулями.
```

а не (если просто убрать вложенные фигурные скобки):

```
int a[2][3] { 1, 2, 4 };
// Остаток заполняется нулями.
```

Сплошное расположение в памяти означает возможность работы с двумерным массивом как с одномерным:

```
void print_matrix
     (int const a[], size_t n, size_t m) {
  for (size_t i = 0, k = 0; i < n; ++i) {
    for (size_t j = 0; j < m; ++j, ++k)
      cout << a[k] << ' ';
    cout << '\n';
  }
}

int main() {
  int arr[3][4] { {1}, {2,3}, {4,5,6} };
  print_matrix(arr[0], 3, 4);
}
```

Обратите внимание, что arr нельзя передать в качестве параметра типа **int**[] (т. е. указателя на **int**), так как он автоматически приводится к **int**(∗)[4] (указателю на подмассив), но не далее.

Многомерный статический массив можно передать в функцию по ссылке или указателю *точно так же*, как одномерный:

```
// Принимает только массив 2×2!
bool equis(int (&a)[2][2]) {
  int s = a[0][0] + a[0][1];
  return s == a[1][0] + a[1][1]
      && s == a[0][0] + a[1][0]
      && s == a[0][1] + a[1][1]
      && s == a[0][0] + a[1][1]
      && s == a[1][0] + a[0][1];
}
```

При передаче по указателю необязательно явно описывать параметр-указатель:

```
float det(float a[2][2]) {
  return a[0][0]*a[1][1]
       - a[1][0]*a[0][1];
}
```

но опустить можно лишь первый размер, поскольку в качестве указателя на массив в таком случае передается указатель на элемент массива, а здесь это — подмассив:

```
// То же самое:
float det(float a[][2]) {
  return a[0][0]*a[1][1]
       - a[1][0]*a[0][1];
}
```

Так происходит из-за того, что компилятору нужно знать шаг в байтах между элементами массива. Чтобы знать этот шаг для самого левого индекса, нужно знать размер в байтах всего подмассива, а значит — его размер.

10.2. Алгебраические структуры

Магма — алгебраическая структура, определяемая как пара $(M, \circ)$, где M — некоторое множество, а $\circ\colon M \times M \to M$ —

бинарная операция, заданная на этом множестве. Такую операцию часто называют «сложением» $+$ или «умножением» $\cdot$.

Полугруппа [*semigroup*] — магма $(M, \circ)$, операция $\circ$ которой *ассоциативна*, т. е. истинно высказывание:

$$(\forall a \in M)(\forall b \in M)(\forall c \in M)\ a \circ (b \circ c) = (a \circ b) \circ c.$$

Например, если операция полугруппы — сложение, то в сумме мы можем расставить скобки произвольным образом, результат от этого не изменится.

Полугруппа $(M, \circ)$ называется *коммутативной*, если истинно высказывание:

$$(\forall a \in M)(\forall b \in M)\ a \circ b = b \circ a.$$

Пример коммутативной полугруппы: $(\mathbb{N}, +)$.

Коммутативность и ассоциативность обеспечивают истинность высказывания «от перестановки мест слагаемых сумма не меняется».

У Придумайте пример коммутативной, но не ассоциативной операции.

Моноид — полугруппа $(M, \circ)$ с *нейтральным элементом*, т. е. истинно высказывание (e — нейтральный элемент):

$$(\exists e \in M)(\forall a \in M)\ e \circ a = a \circ e = a.$$

Моноид удобно записывать как тройку, сразу указывая нейтральный элемент (который в случае умножения принято обозначать 1, а в случае сложения — 0). Например, коммутативным моноидом является $(\mathbb{N}, \cdot, 1)$.

Полугруппу всегда можно расширить до моноида, добавив номинальный нейтральный элемент и доопределив операцию полугруппы. Например, удобно считать 0 элементом $\mathbb{N}$ и говорить о коммутативном моноиде $(\mathbb{N}, +, 0)$.

Группа — моноид $(M, \circ, e)$ такой, что для каждого элемента M существует *обратный элемент*, т. е. истинно высказывание:

$$(\forall a \in M)(\exists b \in M)\ a \circ b = b \circ a = e.$$

Элемент, обратный некоторому элементу a, традиционно обозначают a^{-1}.

Если групповая операция коммутативна, то группу называют *абелевой* или *коммутативной*.

Тройка $(\mathbb{Z}, +, 0)$ является коммутативной группой: для любого $n \in \mathbb{Z}$ обратным будет $(-n)$, так как $n + (-n) = n - n = 0$.

Отметим интересное свойство: $\mathbb{Z}$ можно рассматривать как множество разностей (расстояний со знаком) между элементами $\mathbb{N}$. Ввести их можно формально: для любых натуральных n и m обозначим разность между ними как $+(n-m)$, если $n > m$, как 0, если $n = m$, и как $-(m-n)$, если $n < m$ (разность в натуральных числах). Итак, для случая $n \neq m$ имеем пару *знак*, *натуральное число*. Знак является признаком относительного положения n и m. Множество таких пар, пополненное нулем, и есть множество целых чисел.

У Какой структурой является $(\mathbb{Z}, \cdot, 1)$?

Если переносить эту модель на систему типов C++, то можно было бы ожидать, что разности между целыми числами без знака должны быть числами со знаком. На деле, однако, был выбран иной вариант: разность чисел без знака также является числом без знака. Более того, каждый тип чисел без знака относительно операции сложения образует конечную коммутативную группу (в отличие от натуральных чисел!) благодаря способу обработки переполнения. В то же время из-за того, что переполнение для чисел со знаком возможно, но влечет неопределенное поведение, типы чисел со знаком лишь моделируют бесконечную группу $(\mathbb{Z}, +, 0)$, не реализуя ее семантику в общем случае.

Магма $(\mathbb{R}, \uparrow)$, где $\uparrow$ — операция возведения в степень, не является даже полугруппой, поскольку операция возведения в степень неассоциативна. У данной операции есть *правый нейтрайльный элемент* — 1, но нет *левого нейтрального элемента*.

В случае, если для магмы $(M, \circ)$ уравнение

$$a \circ x = b, \quad a \in M,\ b \in M,$$

всегда имеет единственное решение $x \in M$ (т. е. определено *левое деление* $x = a \backslash b$), то $(M, \circ)$ называется *левой квазигруппой*.

Аналогично, если уравнение

$$y \circ a = b, \quad a \in M, b \in M,$$

всегда имеет единственное решение $y = b/a$ из M (*правое деление*), то $(M, \circ)$ называется *правой квазигруппой*.

Если $(M, \circ)$ является и левой и правой квазигруппой, то она называется *квазигруппой*. Пример: $(\mathbb{Z}, -)$.

У Является ли (левой, правой) квазигруппой магма $(\mathbb{R}, \uparrow)$?

Кольцом называют четверку $(M, \circ, e, *)$ такую, что $(M, \circ, e)$ является коммутативной группой, $(M, *)$ — полугруппой, и выполняется свойство *дистрибутивности*:

$$(\forall a \in M)(\forall b \in M)(\forall c \in M)$$
$$a * (b \circ c) = (a * b) \circ (a * c) \wedge (a \circ b) * c = (a * c) \circ (b * c).$$

Операция $\circ$ обычно называется «сложением», а $*$ — «умножением». Нейтральный элемент e по операции $\circ$ в этом случае можно называть «нулем» и обозначать 0, а возможный нейтральный элемент по операции $*$ — «единицей», обозначать 1.

Если кольцо коммутативно по умножению, то оно называется *коммутативным кольцом*. Если существует некий $m \in M$ такой, что $(M, *, m)$ — моноид, то кольцо называется *кольцом с единицей*.

У Пусть $(M, \circ, e, *)$ — кольцо. Докажите истинность высказывания $(\forall a \in M)\ a * e = e$. (Умножение на нуль дает нуль.)

Тело — кольцо с единицей, в котором все ненулевые элементы образуют группу относительно операции умножения.
Поле — коммутативное кольцо, являющееся телом.

Пример коммутативного кольца с единицей, не являющегося телом (и, следственно, полем): $(\mathbb{Z}, +, 0, \cdot, 1)$. Пример поля: $(\mathbb{Q}, +, 0, \cdot, 1)$ — для любого ненулевого рационального числа существует обратное число.

У Докажите, что $(\mathbb{R}_+, \cdot, *)$ образует поле (определите соответствующие «нуль» и «единицу»). Здесь $\mathbb{R}_+ \triangleq \{\, x \in \mathbb{R} \mid x > 0 \,\}$ — множество положительных действительных чисел. В данном поле операция $\cdot$ (обычное умножение на $\mathbb{R}$) играет роль «сложения». Операция $*$ (играет роль «умножения») определяется следующим образом:

$$a * b \triangleq a^{\ln b}.$$

Если для некоторых ненулевых a и b выполняется $ab = 0$, то такие a и b называются *делителями нуля*. Делители нуля легко ввести на множествах матриц. Например, на множестве числовых матриц 2×2 матрица

$$\begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$$

является делителем нуля относительно стандартной операции матричного умножения.

Коммутативное кольцо с единицей без делителей нуля называют *область целостности* [*integral domain*].

Поле является областью целостности, однако существует промежуточная конструкция, которая должна быть областью целостности, но может не быть полем. Данная конструкция называется *евклидово кольцо* и обладает следующим свойством: можно определить функцию (*евклидову норму*) $E\colon M \setminus \{0\} \mapsto \mathbb{N}_0$ такую, что истинно:

$$(\forall a \in M)(\forall b \in M \setminus \{0\})(\exists q \in M)(\exists r \in M)$$
$$(a = bq + r) \wedge (r = 0 \vee E(r) < E(b)).$$

Таким образом, в евклидовом кольце можно определить деление с остатком, а для вычисления наибольшего общего делителя можно применить алгоритм Евклида.

Отдельный интерес представляют *конечные поля*, обычно обозначаемые $GF(q)$[89], где q — количество элементов поля («порядок поля»). Для выполнения аксиом поля требуется выполнение равенства $q = p^n$, где $n \in \mathbb{N}$, а p («характеристика поля») — простое число. Конечное поле задается порядком с точностью до изоморфизма.

Группа на основе умножения конечного поля является *циклической*, что означает существование *генератора группы*, т. е. такого элемента, умножением которого самого на себя некоторое число раз (иными словами, возведением в натуральную степень) можно получить любой другой элемент группы. Генератор может не быть единственным.

Если q — простое, то с точностью до изоморфизма можно считать, что множество значений $GF(q)$ состоит из целых чисел от 0 до $(q-1)$, сложение — сложение целых по модулю q, умножение — умножение целых по модулю q. К сожалению, в случае иных q конструкция на основе сложения и умножения по модулю (*кольцо вычетов*) не является полем и может содержать делители нуля.

Простейшим примером конечного поля является множество $\{0, 1\}$ (булевский тип), на котором в качестве сложения используется операция *исключающее или* (сложение по модулю 2), а в качестве умножения — конъюнкция.

Увы, типы беззнаковых целых не образуют конечных полей относительно операций сложения и умножения, поскольку имеют непростое количество элементов (степень двойки). Соответственно, для большинства чисел нет обратных по умножению. Тем не менее существуют методы, позволяющие в ряде случаев заменить «честное» деление на константу умножением на другую константу (с некоторой коррекцией и, возможно, обработкой частных случаев). Например, легко проверить, что в 32-битных беззнаковых умножение 3 на 2 863 311 531 дает 1. Однако, чтобы заменить деление на 3 умножением, требует-

[89] От англ. *Galois field* — «поле Галуа», по фамилии фр. математика Э. Галуа (1811–1832).

ся взять 64-битный результат умножения на данную константу (полное умножение, а не по модулю) и сдвинуть его на 33 бита вправо (или верхнюю 32-битную половину результата сдвинуть на 1 бит вправо).

Линейное пространство V (множество *векторов*) над полем F (множество *скаляров*) определяется как коммутативная группа $(V, +)$, для которой определена коммутативная операция умножения $\cdot\colon F \times V \mapsto V$, при условии истинности

$$\begin{aligned}(\forall a \in V)(\forall b \in V)(\forall \alpha \in F)(\forall \beta \in F)\\ (1 \text{ — единица поля } F) \quad 1 \cdot a = a \wedge\\ \alpha(\beta a) = (\alpha\beta)a \wedge\\ \alpha(a + b) = \alpha a + \alpha b \wedge\\ (\alpha + \beta)a = \alpha a + \beta a.\end{aligned}$$

Линейные пространства также называют *векторными*.

Аффинное пространство — пара из пространства векторов V и множества *точек* P такая, что V — линейное пространство над некоторым полем F, заданы операции $+\colon P \times V \mapsto P$ (коммутативная) и $-\colon P \times P \mapsto V$ и истинны

$$\begin{aligned}(\forall a \in V)(\forall b \in V)(\forall p \in P)\\ (p + a) + b = p + (a + b) \wedge\\ (0 \text{ — нуль пространства } V) \quad p + 0 = p.\end{aligned}$$

$$\begin{aligned}(\forall p \in P)(\forall q \in P)(\exists d \in V)\\ p = q + d \text{ (пишем } d = p - q).\end{aligned}$$

Взаимоотношение указателей и целых чисел в C++ напоминают аффинное пространство, где указатели играют роль точек, а целые числа — векторов. (Конечно, множество целых чисел (даже неограниченное) не может считаться линейным пространством.)

На множестве типов также можно ввести ряд операций, в некотором смысле напоминающих операции сложения, умножения (и возведения в степень). Типы, которые получаются

применением таких операций, называются **алгебраическими** [*algebraic data type, ADT*].

Сложение типов (будем обозначать его знаком «|») соответствует операции прямой суммы множеств их значений, т. е. объединению пар (*тип*, *значение*), так что равные значения стоят отдельно друг от друга. Получающийся тип называют *типом-суммой* [*sum type*], или *вариантным типом* [*variant type*], или *типом-объединением* [*union type*].

Значение типа-суммы есть пара (T, v), где T — один из типов суммы, а v — любое значение типа T. Размер множества значений типа-суммы равен сумме размеров множеств значений суммируемых типов.

Например, множество значений типа **bool**|uint8_t[90] состоит из 258 элементов: (**bool**, **false**), (**bool**, **true**), (uint8_t, 0), ..., (uint8_t, 255).

Так как тип **void** имеет пустое множество значений, то разумным отождествить его с нулем и положить **void**|T ≡ T.

Частным случаем типа-суммы может считаться *тип-опция* [*optional*]: None|T, где T — произвольный тип, а тип None — специальный тип, множество значений которого состоит из единственного элемента. Его смысл заключается в представлении ситуации, когда подходящее значение типа T не определено.

Примером такой конструкции еще на уровне языка C является тип указателя: он может быть нулевым (**nullptr** — единственное значение типа nullptr_t) или ненулевым. В последнем случае считается, что он указывает на реальную переменную в памяти. Поэтому тип указателя можно считать типом-опцией на основе соответствующего типа-ссылки (если забыть про арифметику указателей).

Вычитание типов естественным образом определить невозможно.

Умножение типов (будем обозначать его знаком «×») соответствует операции декартова произведения множеств значений типов, т. е. его множеством значений является множе-

[90] Это не C++-код, а условное обозначение.

ство (упорядоченных) кортежей значений всех типов, входящих в произведение. Размер этого множества равен произведению размеров множеств типов, входящих в произведение. Получающийся тип называют *типом-произведением* [*product type*], или *кортежем* [*tuple*], или *записью* [*record*].

Например, тип **bool**×uint8_t имеет 512 значений: (**false**, 0), (**false**, 1), ..., (**false**, 255), (**true**, 0), ..., (**true**, 255).

Умножение на пустой тип дает пустой тип: **void**×T ≡ **void**. У нас нет естественного кандидата на роль единицы по умножению типов, поскольку можно определить много различных типов, каждый из которых имеет лишь одно значение. При необходимости единицу можно ввести формально (например, определить некий тип None или Unit).

Тип отображения A↦R (функциональный тип) из типа A[91] в тип R имеет множество значений[92], количество элементов которого можно записать как $|R|^{|A|}$. Можно представлять себе это как набор таблиц, задающих значения функции для каждого значения аргумента. Данный подход обобщается и на счетные множества аргументов, однако здесь мы попадаем в ситуацию формального существования неопределимых функций, а значит, и непредставимых значений функционального типа.

Например, тип **bool**↦**bool** имеет всего четыре значения, которые на C++ можно записать в виде четырех функций:

```
bool b0(bool) { return false; }
bool b1(bool) { return true; }
bool b2(bool arg) { return arg; }
bool b3(bool arg) { return !arg; }
```

Тип uint8_t↦**bool** имеет уже $2^{256} \approx 1.16 \cdot 10^{77}$ возможных значений.

---

[91] В случае наличия нескольких параметров — это кортеж, в случае отсутствия параметров — это условный тип с одним значением, например, None.

[92] Предполагаются чистые функции, область определения которых совпадает со множеством значений A, такие функции еще называют *полными* [*total*].

У Сколько возможных значений имеет функциональный тип (**bool**×uint16_t)↦(None|**bool**)?

10.3. Гетерогенные типы данных

Под гетерогенными[93] типами в C++ понимаются доступные на уровне языка аналоги типов-произведений и типов-сумм. Для определения типа-произведения используется ключевое слово **struct**[94] («структура»).

Структура состоит из *полей* [*fields*], каждое из которых имеет некоторый тип и может иметь имя. С точки зрения математики структура есть кортеж, т. е. элемент декартова произведения множеств значений ее полей.

```
// Информация о студенте в виде структуры.
struct Student {
  string name[4];
  string group;
  int birth_year, enrollment_year;
};
```

Инициализатор объекта структуры по сути не отличается от инициализатора массива: поля заполняются в порядке их перечисления, незаполненные поля инициализируются по умолчанию (нулями для примитивных типов).

```
Student student {
  { "Smith", "Adam" }, /* вложенный
    инициализатор массива name. */
  "E-170401",
  1999, 2017
};

Student empty {}; // все по умолчанию
```

[93] От др.-греч. *ἑτερογενής* — «разнородный». Противопоставляются гомогенным типам (т. е. «однородным»), к которым относятся примитивные типы и типы массивов.

[94] От лат. *structura* — «построение, фигура речи».

К полю объекта структуры можно обратиться с помощью оператора «точка» [*dot operator*]:

```
student.name[2] = "Walter"; // записать третье имя
cout << student.birth_year; // 1999
student.enrollment_year++;  // поменять на 2018
```

Объекты структур можно копировать с помощью операции присваивания:

```
empty = student; // теперь в empty копия student
```

В отличие от массивов, объекты структур можно передавать в функцию и возвращать из функции по значению. Поэтому, чтобы избежать ненужного копирования, желательно организовывать передачу по ссылке (**const**-ссылке, если объект не требуется изменять в функции).

Аналогично определению перечислимого типа, определение структуры может сразу же определять переменные этого типа (их можно указать между } и ;) и может не включать имя типа, если определены переменные (анонимный тип).

```
// Анонимный тип, две переменных p и q,
// p не инициализирована.
struct { int x, y; } p, q { -1, -1 };
```

Объявление структуры (как и объявление перечислимого типа) не содержит *тела* с объявлением полей:

```
// Определено где-то в другом месте.
struct Info;
```

Как обычно, допустимо произвольное число раз объявлять одну и ту же структуру, но ее определение должно быть единственным в данной единице трансляции.

Невозможно создать переменную объявленной, но не определенной структуры. Однако возможно объявлять ссылки и указатели на объекты таких структур. Реальное обращение по ссылке или указателю (разыменование), естественно, все равно требует наличия выше в коде определения этой структуры.

Структуры, как и прочие типы, можно объявлять и определять внутри функций. Естественно, такие определения видны только внутри тех функций, где они расположены.

Типы можно объявлять и определять внутри структур. Такие типы называются *вложенными* [*nested*]. Их имена доступны через имя структуры, используемое как имя пространства имен[95]. Например, **enum** внутри структуры удобно использовать для определения набора целочисленных констант времени компиляции (имя типа указывать необязательно):

```
struct Star {
  enum Type {
    Supergiant = 1,
    Giant, Subgiant,
    Dwarf, Subdwarf,
    White_dwarf } type;
  enum Color {
    Blue = 1,
    White,
    Yellow,
    Orange,
    Red } color;
  float mass;
};

Star proxima_centauri {
  Star::Dwarf, Star::Red, 0.123f };
```

В случае вложенных структур возможен целый ряд вариантов:

```
struct A {
  struct B { int x, y; };
  struct C { int u, v; } c;
  struct { int w, z; } d;
  struct { int s, t; };
};
```

[95] Впрочем, использовать имя структуры в директиве **using namespace** нельзя.

Здесь A::B и A::C — имена вложенных типов (структур). Две других структуры не имеют имен. Структура B не задает никаких полей внутри A, т. е. это просто определение типа с именем A::B. Структура C помимо определения типа используется сразу как тип поля A::c. Обращение к полям объектов c и d возможно через их имена:

```
A a;
a.c.u = a.c.v = 100;
a.d.w = a.d.z = 0;
A::B b {}; // посторонний объект
```

Однако определение последней структуры не задает ни имени типа, ни имени переменной. В таком случае поля вложенной структуры размечаются внутри внешней структуры и доступны непосредственно:

```
a.s = a.t = -100;
```

Для пользовательских типов (структур и перечислимых типов) возможно определение ряда стандартных операторов. Вообще, с точки зрения C++ применение операторов

```
+  -  *  /  %  &  |  ^  <<  >>  []  ()  ,  ->  ->*
+  -  ~  !  *  &  ++  --  // унарные операции
<  >  ==  !=  <=  >=  &&  ||
=  +=  -=  *=  /=  %=  &=  |=  ^=  <<=  >>=
new  new[]  delete  delete[]
```

эквивалентно вызову функции с именем **operator***on*, где *on* — запись одного из перечисленных выше операторов. С такими функциями можно делать почти все то же самое, что и с обычными функциями. В частности, их можно «перегружать», определяя варианты для своих типов. Ограничения следующие: нельзя менять валентность операций (кроме ()), нельзя вводить определения, все параметры которых принадлежат встроенным типам.

В качестве хорошего примера, демонстрирующего перегрузку операторов и использование понятия аффинного пространства, можно привести определение точек и векторов.

Например, определим тип «двумерный вектор» (Vk2):

```
using FT = float; // field type, скаляр
struct Vk2 { FT x, y; };
```

Для его использования определим набор операторов:

```
/// Сравнение векторов на равенство.
bool operator==(Vk2 const & a, Vk2 const & b) {
  return a.x == b.x && a.y == b.y;
}
/// Сумма векторов.
Vk2 operator+(Vk2 const & a, Vk2 const & b) {
  // используем инициализатор:
  return { a.x + b.x, a.y + b.y };
}
```

C++ не определяет операторы сравнения автоматически. И если мы определили ==, то это еще не значит, что доступна операция !=. Ее, естественно, можно определить как отрицание равенства:

```
bool operator!=(Vk2 const & a, Vk2 const & b) {
  return !(a == b);
}
```

Унарные операторы + и − также определяются независимо от бинарных как самостоятельные функции:

```
/// +вектор возвращает свой аргумент без изменений.
Vk2 operator+(Vk2 const & a) { return a; }
/// Сменить знак всех компонент вектора.
Vk2 operator-(Vk2 const & a) {
  return { -a.x, -a.y };
}
```

Операцию умножения на скаляр приходится определять дважды, поскольку она коммутативна, что, опять же, не предполагается C++ по умолчанию:

```
Vk2 operator*(FT alpha, Vk2 const & a) {
  return { alpha*a.x, alpha*a.y };
}
```

```
Vk2 operator*(Vk2 const & a, FT alpha) {
  return alpha*a; // коммутативная операция
}
```

Для удобства можно определить «обычные» операции текстового ввода-вывода. Операторы << и >> принимают объект потока по ссылке (мы не копируем потоки ввода-вывода) и объект записываемого или читаемого вектора (читаем из потока ввода и записываем в переменную, переданную по ссылке):

```
/// Вывод вектора в поток вывода
/// (просто как пару чисел через пробел).
ostream& operator<<(ostream & os, Vk2 const & v) {
  return os << v.x << ' ' << v.y;
}
/// Ввод вектора из потока ввода (как пары чисел).
istream& operator>>(istream & is, Vk2 & v) {
  return is >> v.x >> v.y;
}
```

Тип «точка на плоскости» (Pt2) определяется аналогично:

```
struct Pt2 { FT x, y; };
/// Начало системы координат (0, 0).
Pt2 const ORIGIN {};
```

Точки нельзя складывать друг с другом, но их можно складывать с векторами, причем эта операция коммутативна:

```
Pt2 operator+(Pt2 const & p, Vk2 const & v) {
  return { p.x + v.x, p.y + v.y };
}

Pt2 operator+(Vk2 const & v, Pt2 const & p) {
  return p + v; // коммутативная операция
}
```

Разность точек дает вектор:

```
Vk2 operator-(Pt2 const & a, Pt2 const & b) {
  return { a.x - b.x, a.y - b.y };
}
```

Радиус-вектор точки можно получить вычитанием ORIGIN.

Необходимо отметить, что C++ также не определяет операции комбинированного присваивания автоматически. Более того, все варианты присваиваний должны определяться как *функции-члены* [*member functions*] (поля структуры можно называть *данными-членами* [*data members*]).

Функция-член объявляется внутри тела структуры и может быть определена в другом месте с явным указанием имени структуры:

```
struct Color {
  float r, g, b;
  float luma() const; // объявление функции-члена
};

// Внешнее определение функции-члена.
float Color::luma() const {
  return .299f*r + .587f*g + .114f*b;
}
```

Вызов функции-члена выполняется также с помощью оператора «точка»:

```
Color const black {};
cout << black.luma() << '\n'; // 0
Color white { 1, 1, 1 };
cout << white.luma() << '\n'; // 1
```

Из тела функции-члена доступны все поля (и функции-члены) объекта структуры, для которого она вызвана. Технически это достигается за счет неявной передачи ей дополнительного параметра — указателя на объект, для которого эта функция вызвана. Получить значение этого указателя можно с помощью ключевого слова **this**. В частности, в примере выше r есть то же самое, что **this**->r. Оператор «стрелка» выполняет обращение к члену структуры через указатель на нее, т. е. **this**->r есть то же самое, что (*__this__).r. Явное обращение через разыменование **this** может понадобиться, если имя затенено локальным определением или если хочется показать, что обращение идет именно к полю данного объекта.

Ключевое слово **const**, записанное после списка параметров в объявлении (и определении) функции-члена, говорит о том, что **this** указывает на константу. Другими словами, данной функции-члену запрещается изменять объект, для которого она вызвана, а значит, нашу функцию Color::luma можно вызвать для константы типа Color.

Функции-члены можно определять прямо внутри структуры. Для таких функций подразумевается спецификатор **inline**.

Операторы присваивания в качестве первого параметра принимают сам объект. Поскольку они могут быть объявлены только как функции-члены, в их заголовке следует указывать только один параметр — то, что стоит справа в операции присваивания. Например, для вектора:

```
struct Vk2 {
  FT x, y;
  // К векторам можно добавлять другие векторы.
  Vk2& operator+=(Vk2 const & a) {
    x += a.x;
    y += a.y;
    return *this; // возвращает ссылку на себя
  }
  // Евклидова длина вектора.
  FT length() const {
    return std::hypot(x, y);
  }
};
```

Функции-члены с модификатором **const** можно вызывать и для временных объектов:

```
Pt2 p, q { 5 };
p = Pt2 { 2, 4 }; // из временного объекта
cout << (p - q).length(); // 5
```

Статический массив внутри объекта структуры является его неотъемлемой частью, и поэтому может быть скопирован (например, передан по значению) в составе этого объекта:

```
struct Data {
  char label[16];
```

```
  float moments[6];
};

// Возвратит массивы внутри объекта по значению:
Data basic_data() { return { {}, {} }; }
```

10.4. Файлы

В данной книге мы не будем рассматривать работу с файлами подробно. Стандартная библиотека C++ предоставляет заголовочный файл **fstream**, в котором определены классы ifstream, ofstream и fstream.

Объекты ifstream предназначены для чтения файлов и могут использоваться так же, как объект стандартного потока ввода cin. Объекты ofstream предназначены для записи файлов и могут использоваться так же, как объект стандартного потока ввода cout. Объекты fstream предоставляют возможности и чтения и записи одновременно.

Открыть файл можно, указав его имя непосредственно в инициализаторе. Проверить, открыт ли файл, можно с помощью функции is_open:

```
#include <fstream>
int main() {
  std::ofstream out("out.txt");
  if (!out.is_open())
    return 1;
  out << "Testing";
  return 0;
}
```

Файл будет закрыт автоматически при уничтожении переменной потока, к которой он привязан. Закрыть файл можно и явно, вызвав функцию close. При необходимости можно открыть файл вызовом функции open (если на этот момент был открыт какой-то файл, то он будет закрыт автоматически).

Поскольку запись в файлы осуществляется через промежуточный буфер в памяти, полезно не держать открытыми фай-

лы, в которые осуществляется запись, неопределенно долго — в случае «падения» программы данные из буфера записаны в файл не будут. Вызовом функции flush можно сбросить данные из буфера в файл, не закрывая его.

По умолчанию файлы открываются в «текстовом режиме». Обычно это означает возможность автоматической замены некоторых последовательностей символов. Например, в строке символ \n это действительно один символ, а в файле — уже два \r\n (в ОС Windows).

В целом, работа с текстовыми файлами может осуществляться точно так же, как со стандартными потоками ввода-вывода. Следующая функция определяет количество и среднюю длину строк заданного файла:

```
struct Line_stats {
  size_t lines;
  double avg_length;
};

Line_stats line_stats(string const & filename) {
  Line_stats ls {};
  ifstream in(filename);
  for (string line; getline(in, line);) {
    ++ls.lines;
    ls.avg_length += line.size();
  }

  if (ls.lines != 0)
    ls.avg_length /= ls.lines;
  return ls;
}
```

Если требуется работа с «двоичными данными» или даже текстовыми данными, но с точностью до байта, то следует открывать файлы в *двоичном режиме*, что достигается с помощью дополнительного параметра ios::bin инициализатора или функции open:

```
ifstream bin_data("assets.dat", ios::bin);
```

Чтение и запись двоичных данных известного размера осуществляется с помощью функций read и write, соответственно, которые принимают адрес массива байт и его размер. Количество успешно проичитанных последней операцией байт возвращает функция gcount:

Пример 10.1. Запись и чтение двоичных данных

```
bool serialize(Vk2 const points[], size_t count) {
  ofstream f("points.bin", ios::bin);
  f.write((char const*)points, sizeof(Vk2) * count);
  return f;
}

// Откуда-то мы знаем предполагаемый размер:
size_t deserialize(Vk2 points[], size_t count) {
  ifstream f("points.bin", ios::bin);
  f.read((char*)points, sizeof(Vk2) * count);
  return f.gcount() / sizeof(Vk2);
}
```

Задания для самопроверки

У Ответьте на следующие вопросы.

- Пусть дана функция

```
void f(int m[3][8]) { /*...*/ }
```

Чему равны значения **sizeof**(m), **sizeof**(m[0]) и std::size(m[0]) внутри ее тела?

- Является ли кольцо квазигруппой?
- Каким аксиомам удовлетворяет любая абелева группа?
- Можно ли назвать (ptrdiff_t, **char***) аффинным пространством?
- Как в C++ определить новый тип, имеющий единственное значение?

• Сколько возможных значений имеет формальный тип **bool**↦(**bool**↦**char**)?

• Сколько возможных значений имеют формальные типы T↦**void** и T↦None? Почему?

• Является ли множество значений типа string↦**bool** счетным?

• Может ли структура не иметь имени? Если да, то каким образом ее можно использовать?

• Как изнутри функции-члена узнать адрес объекта, для которого эта функция-член вызвана?

• Что в примере ниже означает ключевое слово **const**?

```
struct Arg {
  int a, b;
  int max() const {
    return a < b? b: a;
  }
};
```

• Как с помощью **struct** можно сделать ссылку на переменную обычным значением (т. е. изменяемой и допускающей передачу по ссылке)? Попробуйте написать пример.

У Напишите пример определения инициализированного указателя на объект анонимной структуры.

У Проверьте работоспособность кода из примера 10.1. Модифицируйте его таким образом, чтобы размер массива сохранялся в файле перед содержимым самого массива и чтобы функция deserialize не принимала размер массива, а считывала его из файла.

Глава 11

Объекты

11.1. Многомерные динамические массивы

Рассмотрим способы организации многомерных динамических массивов на примере двумерных массивов.

К рассмотрению предлагается три способа:

1. Динамический массив динамических массивов.
2. Динамический упакованный массив.
3. Комбинация первых двух.

В случае статических массивов массив организуется как массив массивов, но хранится как упакованный. Это возможно благодаря тому, что нет нужды хранить информацию об отдельных подмассивах, поскольку она «зашита» в тип переменной массива на этапе компиляции.

Способ 1

Структура: динамический массив указателей на (динамические) массивы меньшего порядка (например, в случае матрицы — строки). Преимуществом такой конструкции являет-

ся синтаксическая простота обращения к элементу через стандартный оператор [], например, m[i][j] — элемент матрицы m в строке i, столбце j.

Обозначим тип элемента через ET:

```
using ET = int; // «Element Type»
```

Создание такой матрицы:

```
ET ** new_mtx(int rows, int cols) {
  // Создать массив указателей на строки.
  auto m = new ET*[rows];
  // Создать строки.
  for (int row = 0; row < rows; ++row)
    m[row] = new ET[cols];
  return m;
}
```

У Перепишите функцию new_mtx таким образом, чтобы избежать возможного бросания исключения оператором **new** и потенциальной утечки памяти.

Соответствующее удаление:

```
void delete_mtx(ET ** m, int rows) {
  // Нулевой указатель не должен приводить к ошибке.
  if (!m) return;
  // Удалить каждую строку.
  for (int row = 0; row < rows; ++row)
    delete[] m[row];
  // Удалить массив указателей на строки.
  delete[] m;
}
```

Удалению не нужно знать второй размер (в случае аналогичного n-мерного массива — последний размер), так как оператор **delete**[] не нуждается в указании размера удаляемого массива.

В качестве примера простой функции, обрабатывающей объект матрицы, приведем функцию fill_mtx, выполняющую заполнение уже созданной в памяти матрицы одним и тем же значением:

```
void fill_mtx(ET ** m, int rows, int cols, ET val) {
  for (int row = 0; row < rows; ++row) {
    auto const r = m[row]; // указатель на строку
    for (int col = 0; col < cols; ++col)
      r[col] = val;
  }
}
```

Мы могли написать во внутреннем цикле

```
m[row][col] = val;
```

но с точки зрения производительности лучше заранее извлечь указатель на строку, коим является m[row], в локальную переменную.

Недостатком данного способа организации многомерного массива является множество выделений и освобождений динамической памяти и возможная «разбросанность» подмассивов в памяти, что отрицательно сказывается на эффективности кэширования и предзагрузки данных процессором при обходе матрицы.

Преимуществом данного способа является относительная гибкость: можно, например, заменять или переставлять подмассивы, не затрагивая весь массив (достаточно изменить соответствующие указатели головного массива). Можно даже создавать подмассивы разной длины — *рваный массив* [*jagged array, ragged array*].

Способ 2

Способ 2 предполагает запись всего содержимого многомерного массива в один сплошной одномерный массив. Подмассивы укладываются в памяти друг за другом. Для обращения к элементам многомерные индексы требуется переводить в одномерные. То есть явно делать то, что делает компилятор при работе со статическими многомерными массивами.

Массив с размерностями $(d_0, d_1, \ldots, d_{r-1})$ содержит всего $d_0 \cdot d_1 \cdots d_{r-1}$ элементов. Количество размерностей r называют *рангом* [*rank*] массива. При укладке их подряд в памяти

в духе статического многомерного массива получаем следующую формулу приведения r-мерного (векторного) индекса $(i_0, i_1, \ldots, i_{r-1})$ к одномерному (скалярному) индексу I в соответствующем одномерном массиве:

$$I = i_{r-1} + \sum_{j=0}^{r-2} \left(i_j \prod_{k=j+1}^{r-1} d_k \right).$$

Следует обратить внимание, что данная формула не требует выполнения повторных перемножений при суммировании, ее вычислительная сложность линейна по r. Это можно реализовать, например, так:

```
size_t scalar_index // ранг r > 0!
    (int r, size_t d[], size_t i[]) {
  size_t I = i[0];
  for (int s = 1; s < r; ++s)
    I = I*d[s] + i[s];
  return I;
}
```

В случае трехмерного массива она обретает вид:

$$I = i_2 + i_1 d_2 + i_0 d_1 d_2 = i_2 + d_2(i_1 + d_1 i_0).$$

В случае же двумерного массива она весьма проста:

$$I = i_1 + i_0 d_1.$$

Приведем пример, аналогичный примеру для предыдущего способа. Создание матрицы:

```
ET * new_mtx(int rows, int cols) {
  // Создать массив, содержащий все элементы.
  return new ET[size_t(rows) * cols];
}
```

Данная реализация упрощена: мы игнорируем возможность переполнения при умножении rows на cols[96], что может повлечь труднообнаружимые ошибки в коде.

Соответствующее удаление элементарно:

```
void delete_mtx(ET * m) {
  delete[] m;
}
```

Реализация fill_mtx:

```
// Заполнить матрицу константой.
void fill_mtx(ET * m, int rows, int cols, ET val) {
  // Так как все элементы одинаковые и идут подряд,
  // можно организовать одномерный обход.
  auto const sz = size_t(rows) * cols;
  for (size_t i = 0; i < sz; ++i)
    m[i] = val;
}
```

Вывод матрицы на экран (пример двойного цикла):

```
void print_mtx(ET const * m, int rows, int cols) {
  for (int row = 0; row < rows; ++row) {
    for (int col = 0; col < cols; ++col)
      cout << ' ' << m[col];
    cout << '\n';
    m += cols; // к следующей строке
  }
}
```

Операцию обращения по двумерному индексу можно выразить с помощью функции (но при последовательном обходе массива использовать эту формулу не следует):

```
ET & mtx_el(ET * m, int cols, int row, int col) {
  return m[row*cols + col];
}
```

[96] В случае 32-битного **int** и 64-битного size_t (что типично для современных 64-битных систем) здесь переполнение невозможно, если переданные размеры не отрицательны.

Преимуществами способа 2 являются:

- удобство кодирования и в среднем большее быстродействие операций, выполняемых над массивом целиком;
- минимизация числа операций выделения и освобождения памяти;
- минимизация затрат памяти (нет вспомогательного массива указателей, нет заголовков блоков динамической памяти на каждый подмассив).

Недостатки:

- обратная сторона минимизации числа операций: выделение сразу большого куска памяти может производиться медленно или быть вовсе невозможным из-за фрагментации кучи;
- простые операции, вроде перестановки строк, невозможно выполнить простой манипуляцией указателями: необходимо либо явно обменивать все элементы строк, либо применять промежуточное преобразование индексов, либо создавать новый измененный массив;
- наконец, недоступен привычный синтаксис обращения к элементу вида m[i][j]. Через [] обращение идет по скалярному индексу. (Этот последний недостаток поправим благодаря возможности перегрузки операторов.)

Способ 3

Данный способ является комбинацией двух предыдущих и удобен в случае двумерных массивов. Память выделяется сразу на все элементы массива (первый блок) и отдельно на головной массив с указателями, которые инициализируются вычислением смещений подмассивов (второй блок). В примере ниже указатель на массив-хранилище записывается «перед» первым

элементом головного массива, чтобы можно было корректно удалить хранилище, не опираясь на, возможно, измененные адреса подмассивов.

Обращаться к элементам такой матрицы можно и так, как в способе 1, и так, как в способе 2.

Создание матрицы:

```
ET ** new_mtx(int rows, int cols) {
  // Создать массив указателей на строки матрицы.
  auto m = new ET*[rows + 1];
  // Создать массив, содержащий все элементы.
  m[1] = m[0] = new ET[size_t(rows) * cols];
  // Заполнить указатели на строки.
  for (int i = 2; i <= rows; ++i)
    m[i] = m[i-1] + cols;
  return m + 1;
}
```

(Опять для простоты игнорируем возможность переполнения при перемножении rows на cols.)

Соответствующее удаление:

```
void delete_mtx(ET ** m) {
  // Удалить массив элементов.
  --m;
  delete[] m[0];
  // Удалить массив указателей.
  delete[] m;
}
```

Реализация fill_mtx:

```
// Заполнить матрицу константой.
void fill_mtx(ET ** m, int rows, int cols, ET val) {
  // Так как все элементы одинаковые и идут подряд,
  // можно организовать обход как линейного массива.
  auto const sz = size_t(rows) * cols;
  auto p = m[-1]; // указатель на хранилище
  for (size_t i = 0; i < sz; ++i)
    p[i] = val;
}
```

По сравнению со способом 1 мы получаем ряд преимуществ способа 2, связанных с хранением элементов упакованными в один массив (но теперь все же тратим дополнительную память на массив указателей на строки).

В отличие от способа 2 мы можем обращаться к элементам, как в способе 1 и передавать указатель функциям, читающим массивы в формате способа 1. И даже более того: мы можем потерять любые указатели на строки (например, «удалить» столбцы слева, сдвинув указатели вправо), это не приведет к утечке памяти, поскольку указатель на хранилище записан в отдельный, скрытый элемент в массиве указателей.

11.2. Объекты

На примере операций мы могли видеть, что синтаксис применения различных вспомогательных функций требовал передавать описание матрицы из одного-трех элементов. Удобнее разместить все данные матрицы в одном месте, определив соответствующую структуру (далее на примере способа 1):

```
struct Matrix {
  int rows, cols;
  ET ** data;
};
```

тогда можно переделать все функции, чтобы они принимали ссылку на объект Matrix.

Но C++ позволяет сделать управление ресурсом (в данном случае — памятью) более простым для пользователя. Для этого следует рассматривать матрицу как *объект*[97].

Объект [*object*] — структура данных и набор связанных действий (функций-членов, также называемых *методы*), самостоятельно управляющая своим состоянием.

[97] Теперь данное слово употребляется как термин и основное понятие *объектно-ориентированного программирования*.

Данное определение акцентирует внимание на ответственности: объект несет ответственность за то, чтобы его состояние было корректно на протяжении его жизни. Этим объект отличается от просто значения с технической точки зрения.

С концептуальной точки зрения объект — отражение некоторой конкретной сущности. Именно поэтому имеет смысл говорить о его времени жизни[98].

Время жизни объекта [*object lifetime*] — временной отрезок, в течение которого объект считается существующим, и им можно пользоваться (обращаться к полям и функциям-членам). Начинается после успешного завершения работы *конструктора* и завершается в момент вызова *деструктора*.

Конструкторы и деструкторы — специальные функции, отвечающие за инициализацию и корректное завершение существования объектов.

Жизненный цикл объекта состоит из следующих стадий:

1. Выделение памяти под объект.
2. Вызов конструктора (может быть пропущен).
3. Время жизни объекта (корректное состояние объекта).
4. Вызов деструктора (может быть пропущен).
5. Освобождение памяти, выделявшейся под объект.

У примитивных типов вроде **int**, **float** или **char*** нет ни конструктора[99], ни деструктора. Соответственно, они по умолчанию не инициализируются (при статическом размещении они имеют начальное нулевое значение, поскольку вся статическая память обнуляется на старте программы).

Другое дело, например, std :: string. Этот тип определяет и набор конструкторов и деструктор, освобождающий память,

[98] Действительно, какое время жизни у числа 10 или функции sin? Это — примеры абстрактных сущностей.

[99] Точнее, нет *конструктора по умолчанию*.

занятую строкой. Поэтому значения std::string являются полноценными объектами.

Типы объектов принято называть *классами*. В C++ есть ключевое слово **class**, смысл которого весьма близок **struct**.

Конструктор [*constructor*] — специальная функция, определяемая классом и выполняемая при создании нового объекта класса (*экземпляра* [*instance*]). Конструкторов может быть много, но в процессе создания объекта обязательно вызывается один из них.

В качестве имени конструктора (как функции) используется имя типа, для которого этот конструктор определен. Конструктор не имеет возвращаемого значения (соответственно, тип возвращаемого значения не пишется).

После успешного завершения конструктора объект считается **полностью созданным** [*completely constructed*] и может использоваться.

Если выполнение конструктора было прервано бросанием исключения, то объект считается **частично созданным** [*partially constructed*]. Для всех полей частично созданного объекта, для которых уже были выполнены конструкторы, автоматически вызываются деструкторы (в порядке, обратном порядку вызова конструкторов). Деструктор для частично созданного объекта не вызывается, поскольку деструктор предполагает корректное состояние объекта.

Конструктор по умолчанию [*default constructor*] не принимает параметров. Обычно генерируется компилятором автоматически, если пользователь не определил ни одного конструктора, и каждое поле либо имеет свой конструктор по умолчанию, либо не требует инициализации.

В случае нашего типа Matrix конструктор по умолчанию ничего не делает, поскольку все поля принадлежат примитивным типам. Мы можем определить конструктор по умолчанию, заполняющий все поля нулями (пустая матрица):

```
struct Matrix {
  int rows, cols;
```

```
  ET ** data;
  // Конструктор по умолчанию.
  Matrix() {
    rows = 0;
    cols = 0;
    data = nullptr;
  }
};
```

Конструктор может явно вызывать конструкторы полей или другой конструктор объекта с помощью синтаксиса, называемого **список инициализации** [*initialization list*]. Эти конструкторы выполняются до выполнения тела вызванного конструктора объекта, причем их порядок строго задан определением класса (в порядке перечисления полей сверху вниз). В конце выполняется тело конструктора (все поля, для которых есть конструкторы, к этому моменту уже инициализированы).

```
struct Matrix {
  int rows, cols;
  ET ** data;
  // Конструктор по умолчанию.
  Matrix()
    : rows(0), cols(0), data(nullptr)
  { /* пустое тело */ }
};
```

Список инициализации указывается после двоеточия между списком параметров конструктора и телом конструктора. Список инициализации состоит из перечисления имен полей со списками параметров, которые мы хотим передать соответствующим конструкторам. Порядок перечисления полей внутри списка инициализации не влияет на реальный порядок вызовов их конструкторов.

Список инициализации не обязан перечислять все поля — для тех полей, которые не были указаны, будет выполнена инициализация по умолчанию.

Вместо вызовов конструкторов полей список инициализации может содержать ровно один вызов другого собственного конструктора. Это позволяет избежать дублирования кода или введения вспомогательных функций.

Без списка инициализации возникла бы проблема с использованием в качестве типов полей таких классов, у которых нет конструкторов по умолчанию, так как их конструкторы требуют передачи конкретных параметров, которые никак не передать из тела конструктора объекта.

Теперь добавим конструктор, принимающий размеры матрицы (замена функции new_mtx):

```
Matrix(int rows, int cols)
  : rows(rows), cols(cols) {
  // Создать массив указателей на строки.
  data = new ET*[rows] {};
  // Создать строки.
  for (int row = 0; row < rows; ++row)
    data[row] = new ET[cols];
}
```

Деструктор [*destructor*] — специальная функция-член класса, не принимающая параметров и ничего не возвращающая. Деструктор вызывается в процессе уничтожения объекта, если объект ранее был полностью создан. Класс может определить только один деструктор. После тела деструктора вызываются деструкторы полей в порядке их перечисления в коде снизу вверх. После вызова деструктора объект считается переставшим существовать, и его использование влечет неопределенное поведение. Деструктор не освобождает память, занимаемую самим объектом, — это делается уже после завершения деструктора. Имя деструктора состоит из знака ~ и имени типа.

Нельзя взять адрес конструктора или деструктора.

Определим деструктор матрицы (замена delete_mtx):

```
~Matrix() {
  if (data) { // Удалить каждую строку.
    for (int row = 0; row < rows; ++row)
```

```
      delete[] data[row];
    // Удалить массив указателей на строки.
    delete[] data;
  }
}
```

Функцию fill_mtx теперь можно сделать функцией-членом fill. Так как все данные матрицы доступны из объекта, то fill достаточно принимать только один параметр — значение, которым заполняется матрица:

```
void fill(ET val) {
  for (int row = 0; row < rows; ++row) {
    auto const r = data[row]; // указатель на строку
    for (int col = 0; col < cols; ++col)
      r[col] = val;
  }
}
```

Для удобства обращения к элементам матрицы (чтобы не писать .data) определим оператор []:

```
ET * operator[](int row) {
  return data[row];
}
```

Вывод матрицы в текстовый поток вывода удобно реализовать, определив оператор <<:

```
ostream & operator<<
    (ostream & os, Matrix const & mtx) {
  auto const rows = mtx.rows, cols = mtx.cols;
  for (int row = 0; row < rows; ++row) {
    auto const r = mtx.data[row];
    for (int col = 0; col < cols; ++col)
      os << ' ' << r[col];
    os << '\n';
  }
  return os;
}
```

Теперь можно написать простенькую программу, демонстрирующую работу с матрицей:

```
int main() {
  int const ROWS = 4, COLS = 5;
  Matrix m(ROWS, COLS);
  m.fill(0);
  m[ROWS/2][COLS/2] = 1;
  cout << m;
  return 0;
}
```

Деструктор m будет вызван автоматически, когда закончится область видимости переменной m, т. е. придет момент освобождения памяти, занятой этой переменной.

Как видно, передать параметры конструктору можно, перечислив их в круглых скобках после имени переменной. Запись вида T(args), где T — имя типа, есть вызов конструктора T с передачей ему аргументов args. Результат такого вызова есть объект типа T.

Данная схема обобщена и на примитивные типы и структуры без конструкторов. Соответствующие функции называются *псевдоконструкторами*. Так, size_t(−1) это, по сути, создание объекта типа size_t из значения −1 (реализуется как приведение типа). А выражение size_t() есть то же, что size_t(0).

Для структур может быть доступен псевдоконструктор по умолчанию, заполняющий все поля примитивных типов нулями. Если же структура включает поле (или подполе), для которого определен настоящий конструктор по умолчанию, то для этой структуры компилятор также порождает настоящий конструктор по умолчанию, который вызывает конструкторы тех полей, для которых они определены.

Напишем новую реализацию способа 2. Это позволит нам сделать его внешне таким же удобным, как для способа 1.

```
struct Matrix {
  int rows, cols;
  ET * data;
  // Конструктор по умолчанию (пустая матрица).
  Matrix()
    : rows(0), cols(0), data(nullptr) {}
```

```
  // Конструктор матрицы заданных размеров.
  Matrix(int rows, int cols)
    : rows(rows), cols(cols),
      data(new ET[size_t(rows) * cols]) {}
  // Деструктор.
  ~Matrix() { delete[] data; }
  // Заполнить матрицу константой.
  void fill(ET val) {
    auto const sz = size_t(rows) * cols;
    auto const p = data;
    for (size_t i = 0; i < sz; ++i)
      p[i] = val;
  }
  // Обратиться к строке матрицы.
  ET * operator[](int row) {
    return data + row*cols;
  }
  ET const * operator[](int row) const {
    return data + row*cols;
  }
};
```

Теперь можно обращаться с Matrix абсолютно так же, как в предыдущем случае:

```
  Matrix m(4, 5);
  m.fill(0);
  m[2][2] = 1;
```

11.3. Правило трех

Что произойдет при выполнении следующего кода?

```
int main() {
  Matrix m(10, 10);
  Matrix z = m;
}
```

Поля m будут скопированы по значению в поля z. Деструкторы вызываются в порядке, обратном порядку вызова конструкторов, поэтому сначала будет вызван деструктор z, который благополучно удалит массив, указатель на который все еще хранится в m. После чего деструктор m попробует удалить тот же массив. Иными словами, у нас двойное удаление, т. е. неопределенное поведение. Это серьезная ошибка.

Чтобы исправить ситуацию, надо самостоятельно определить *копирующий конструктор* [*copy constructor*]. Такой конструктор принимает ссылку на объект своего класса:

```
Matrix(Matrix const & other)
  : Matrix(other.rows, other.cols) {
  // Скопировать элементы.
  for (int row = 0; row < rows; ++row) {
    auto const s = other.data[row];
    auto const d = data[row];
    for (int col = 0; col < cols; ++col)
      d[col] = s[col];
  }
}
```

В списке инициализации мы вызываем уже имеющийся конструктор, чтобы выделить хранилище. Затем поэлементно копируем матрицу. Теперь ошибки с двойным удалением не будет: m и z будут содержать одно и то же, но хранилища будут разными.

Но что произойдет при выполнении следующего кода?

```
int main() {
  Matrix m(10, 10);
  Matrix z(20, 20);
  z = m;
}
```

Обе матрицы были созданы независимо. При присваивании опять выполняется копирование всех полей. Итак, мы имеем сразу две ошибки: утечку памяти из-за потери указателя на хранилище z и затем опять двойное удаление хранилища m!

Для того чтобы исправить ситуацию, следует определить *копирующий оператор присваивания* [*copy assignment operator*]. Этот оператор также принимает ссылку на объект своего класса, но в отличие от конструктора мы имеем уже инициализированный объект, хранилище которого (возможно) следует удалить. Данный оператор можно реализовать через копирующий конструктор идиоматической конструкцией на основе функции swap (стандартная версия определена в заголовочном файле **utility**):

```
// Обменять содержимое двух матриц.
void swap(Matrix & other) {
  std::swap(rows, other.rows);
  std::swap(cols, other.cols);
  std::swap(data, other.data);
}
// Копирующий оператор присваивания.
Matrix & operator=(Matrix const & other) {
  Matrix(other).swap(*this);
  return *this;
}
```

Здесь происходит следующее: мы создаем временный объект матрицы как копию other (используя определенный выше конструктор), затем обмениваем его содержимое с содержимым нашего объекта (**this**). Теперь наш объект хранит копию, а его старое хранилище управляется временным объектом, который уничтожается деструктором по достижении ; (конец времени жизни временного объекта).

Итак, если мы сами не определим копирующий конструктор и копирующий оператор присваивания, то компилятор может это сделать за нас, однако не факт, что результат нам понравится. Эти автоматически определяемые копирующие конструктор и оператор присваивания называются *тривиальный копирующий конструктор* и *тривиальный копирующий оператор присваивания* (аналогично тривиальному конструктору

по умолчанию). Их «тривиальность»[100] заключается в том, что они просто копируют двоичное представление полей как куска памяти.

Правило трех [*the Rule of Three*]. Если требуется определить что-то из следующего списка:

1. Деструктор.
2. Копирующий конструктор.
3. Копирующий оператор присваивания.

То следует определить **все три**.

11.4. Сокрытие данных

Сокрытие данных или **инкапсуляция**[101] [*encapsulation*] — явное разделение доступа к состоянию объекта между им самим и «внешним миром».

Инкапсуляцию можно считать прямым приложением концепции модульного программирования к объектам. С этой точки зрения объекты можно считать «модулями», у которых есть доступная извне интерфейсная часть и скрытая реализация, доступ к которой осуществляется через интерфейсную часть.

В C++ основным инструментом такого разделения является управление доступом к членам структуры (класса) с помощью ключевых слов **public** (общедоступная, открытая часть) и **private** (скрытая часть). «Сокрытие» обеспечивается ошибкой компиляции при попытке обратиться к private-членам не из функции-члена класса.

---

[100] Слово *тривиальный* происходит от лат. *trivium* (*tri-* — «три-» и *via* — «дорога»). Под «тремя дорогами» понимались грамматика, логика и риторика, с которых начиналось обучение в средневековых университетах.

[101] От лат. *in* — «в», *capsula* — «ящичек, шкатулка» (*capsa* — «вместилище», *capio* — «беру, вмещаю»).

Данные ключевые слова вводят секции внутри структуры или класса:

```
struct A {
public:
  // Открытая часть...
  int a;
  int b;
private:
  // Закрытая часть...
  int c;
  int d;
};
```

Таких секций может быть произвольное количество.

Теперь о разнице между **struct** и **class**. Разница заключается в режиме доступа по умолчанию:

```
struct A {
  // ...
```

есть то же, что

```
struct A {
public:
  // ...
```

В то же время

```
class B {
  // ...
```

есть то же, что

```
class B {
private:
  // ...
```

То есть члены «структур» по умолчанию открыты, а члены «классов» — скрыты. Структуры можно считать классами.

Разница носит стилистический характер: как **struct** принято определять типы, не предполагающие управление состоянием или используемые в стиле C или для работы с функциями, объявленными как **extern** "C".

Напротив, как **class** принято определять типы с нетривиальной семантикой, наличием закрытой части, используемые только в C++-коде. Открытую часть рекомендуется размещать в начале определения класса.

Переделаем наши реализации Matrix.

Определения функций можно вынести из тела структуры или класса. Исторически было принято разделять объявление и определение таким образом, что определение класса с объявлениями функций-членов помещалось в заголовочный файл, а определения этих функций-членов — в отдельный файл исходного кода. Функция-член, определенная в определении класса, автоматически получает спецификатор **inline**.

```
class Matrix {
public:
  // Деструктор.
  ~Matrix();
  // Конструктор по умолчанию.
  Matrix();
  // Конструктор матрицы заданных размеров.
  Matrix(int rows, int cols);
  // Копирующий конструктор.
  Matrix(Matrix const &);
  // Копирующий оператор присваивания.
  Matrix & operator=(Matrix const &);
  // Заполнить матрицу константой.
  void fill(ET val);
  // Обменять содержимое двух матриц.
  void swap(Matrix &);
  // Сколько строк?
  int rows() const { return _rows; }
  // Сколько столбцов?
  int cols() const { return _cols; }
  // Доступ к строке по ее индексу.
  ET * operator[](int row) // inline
  { return _data[row]; }
  ET const * operator[](int row) const // inline
  { return _data[row]; }
private:
```

```
  // Сокрытые данные объекта.
  int _rows, _cols;
  ET ** _data;
};
```

При записи внешнего определения функции-члена класса нужно поставить имя класса (через :: аналогично пространствам имен) перед именем функции. Например:

```
Matrix & Matrix::operator=(Matrix const & other) {
  Matrix(other).swap(*this);
  return *this;
}
```

Теперь обратимся к конструктору, собственно выделяющему память. В нем осталась одна неприятная ошибка.

```
Matrix::Matrix(int rows, int cols)
  : _rows(rows), _cols(cols) {
  // Создать массив указателей на строки.
  _data = new ET*[rows] {};
  // Создать строки.
  for (int row = 0; row < rows; ++row)
    _data[row] = new ET[cols];
    // ^^^ что если здесь будет исключение?
}
```

Что если мы не сможем выполнить цикл полностью? Объект будет частично создан, а значит, деструктор не будет вызван. А значит, уже выделенная к этому моменту память не будет освобождена!

Данную ситуацию можно исправить несколькими способами. Рассмотрим способ, который использует форму оператора **new**, возвращающую в случае невозможности выделения памяти нулевой указатель — **new**(nothrow). В случае невозможности выделения памяти освободим всю память, выделенную к данному моменту, и бросим исключение bad_alloc (ошибка выделения памяти):

```
Matrix::Matrix(int rows, int cols)
  : _rows(rows), _cols(cols) {
  // Создать массив указателей на строки.
  _data = new ET*[rows];
  // Создать строки.
  for (int row = 0; row < rows; ++row) {
    auto const items = new(nothrow) ET[cols];
    _data[row] = items;
    if (!items) {
      // Ошибка выделения памяти.
      for (int i = 0; i < row; ++i)
        delete[] _data[i];
      delete[] _data;
      // Исключение «не удалось выделить память».
      throw bad_alloc();
    }
  }
}
```

Однако есть более предпочтительный способ, который заключается в применении **объектной декомпозиции** — разделения состояния объекта на набор объектов, которые сами управляют своим состоянием. В данном случае напрашивается создание объекта, управляющего памятью строки матрицы. Для этого введем два новых класса, очень похожих друг на друга (позже мы от них избавимся):

```
// Класс, управляющий строкой.
class Row {
public:
  // Деструктор.
  ~Row() { delete[] _data; }
  // Конструктор по умолчанию.
  Row(): _data(nullptr) {}
  // Конструктор строки заданного размера.
  Row(int cols): _data(new ET[cols]) {}
  // Копирование запрещено!
  Row(Row const&) = delete;
  Row& operator=(Row const&) = delete;
  // Доступ к строке.
```

```
  ET * data() { return _data; }
  ET const * data() const { return _data; }
  // Обмен с другим объектом.
  void swap(Row & other) {
    std::swap(_data, other._data);
  }
private:
  ET * _data;
};

// Класс, управляющий массивом строк.
class Rows {
public:
  // Деструктор.
  ~Rows() { delete[] _data; }
  // Конструктор по умолчанию.
  Rows(): _data(nullptr) {}
  // Конструктор строки заданного размера.
  Rows(int rows): _data(new Row[rows]) {}
  // Копирование запрещено!
  Rows(Row const&) = delete;
  Rows& operator=(Rows const&) = delete;
  // Доступ к строке.
  Row * data() { return _data; }
  Row const * data() const { return _data; }
  // Обмен с другим объектом.
  void swap(Rows & other) {
    std::swap(_data, other._data);
  }
private:
  Row * _data;
};
```

Запись вида

```
  Row(Row const&) = delete;
  Row& operator=(Row const&) = delete;
```

где объявление функции содержит элемент = **delete**, позволяет запретить автоматическое порождение данной функции. Таким образом, мы запрещаем копирование объектов Row.

Теперь в Matrix заменим ET ** _data на Rows _data. Деструктор Matrix можно вовсе убрать, поскольку строки будут удалены деструктором Rows.

А конструктор, принимающий размеры, уже не требует ручного освобождения памяти в случае ошибки, поскольку это будет сделано автоматически:

```
Matrix::Matrix(int rows, int cols)
  : _rows(rows), _cols(cols), _data(rows) {
  // Создать строки.
  for (int row = 0; row < rows; ++row)
    Row(cols).swap(_data.data()[row]);
}
```

11.5. Static-члены класса

Определение типа **struct** или **class** может содержать члены-данные и члены-функции, объявленные **static**. Смысл этого модификатора в контексте классов отличается от его смысла в контексте единиц трансляции и заключается в том, чтобы определить член класса, который не является членом объекта.

Иными словами, «**static**-поле» есть своеобразная глобальная переменная, создаваемая при создании переменных единицы трансляции, в которой она определена, но размещенная в области видимости класса (с учетом возможного ограничения доступа).

Функция класса, определенная как **static**, есть обыкновенная функция, помещенная в область видимости класса и вызываемая через имя класса, а не через объект класса (естественно, такая функция не получает указатель **this**):

```
class Thing {
  // Счетчик текущего числа объектов Thing:
  static size_t _count = 0;
  // доступен только членам-функциям.
public:
  Thing() { ++_count; }
```

```
  ~Thing() { --_count; }
  static size_t count() {
    return _count;
  }
};

int main() {
  cout << Thing::count();    // 0
  Thing a;
  {
    Thing b;
    cout << Thing::count(); // 2
  }
  cout << Thing::count();    // 1
}
```

Члены-данные с модификатором **static** целочисленного типа или объявленные с ключевым словом **inline** могут быть определены прямо в определении класса. Прочие должны быть определены отдельно и будут размещены в памяти той единицы трансляции, в которой они определены (это может быть удобно, например, если требуется сложный инициализатор). В этом случае ключевое слово **static** в определении не пишется.

11.6. Ссылки на временные значения

Рассмотрим следующую функцию:

```
// Создать единичную матрицу.
Matrix identity(int n) {
  Matrix id(n, n);
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      id[i][j] = i == j;
  return id;
}
```

Вызов подобной функции в инициализации переменной

```
auto E = identity(4);
```

подразумевает выполнение следующих действий:

- выделить память в стеке под временный объект Matrix;
- вызвать функцию identity, в которой создать локальную переменную id;
- в инструкции **return** инициализировать временный объект Matrix с помощью копирующего конструктора копией локальной переменной id;
- уничтожить id (деструктор);
- вернуться из вызова в точку определения переменной E;
- инициализировать объект E копией временного объекта Matrix (еще один вызов копирующего конструктора);
- уничтожить временный объект Matrix (еще раз вызвать деструктор).

Итого, два копирования матрицы и два уничтожения копий. На деле довольно давно существует применяемый компиляторами прием, называемый *оптимизация возвращаемого (именованного) значения* [*(N)RVO, (named) return value optimization*][102], заключающийся в том, что в качестве возвращаемой переменной подставляется переменная, инициализируемая этим значением в точке вызова, чем можно полностью избежать копирования и удаления промежуточных объектов.

Однако, во-первых, компилятор не обязан делать (N)RVO. Во-вторых, компилятор не стал бы делать (N)RVO, если конструктор копии или деструктор содержат наблюдаемые побочные эффекты (например, мы считаем количество созданных

[102] Вариант «именованного» (NRVO) как раз относится к нашему примеру и проще в реализации, чем «просто» RVO — вариант возврата временного значения (результата выражения в **return**), который, по сути, требует рекурсивного применения NRVO к цепочке возвратов.

объектов с помощью внешнего счетчика или выполняем операции ввода-вывода), так как это нарушило бы предполагаемую семантику программы.

В C++17 данную проблему решили устранить, постановив, что в ситуации, подобной выше (инициализация временным значением), компилятор обязан выполнить прямую инициализацию итогового объекта, устранив промежуточные копирования («copy elision») даже в том случае, если конструктор копии или деструктор содержат наблюдаемые побочные эффекты.

Рассмотрим другой случай.

```
Matrix M(10, 10);
// ...
M = identity(10);
```

Здесь уже нет инициализации: объект M был создан до присваивания. Устранение промежуточного копирования не может устранить вызов копирующего оператора присваивания, а устраняет только копирование id во временный объект Matrix. Таким образом, здесь у нас остается одно копирование. Мы бы могли его избежать, например, так:

```
identity(10).swap(M);
```

Есть и ряд других случаев, когда технически копирование могло бы быть убрано, но семантика программы его требует. Для решения подобных проблем в C++11 было введено понятие *ссылка на вре́менное значение* [*temporary value reference*], называемая также rvalue reference.

Вводится новый ссылочный тип T&& — ссылка на временное значение типа T. Итак, пусть T не содержит модификатора **const**. Следующая таблица описывает способ осуществления связывания параметра (левый столбец) и аргумента (верхняя строчка):

Парам. \ арг.	T, T&&	T&	T **const**&
T	копия/перем.	копия	копия
T&&	по ссылке	—	—
T&	—	по ссылке	—
T **const**&	по ссылке	по ссылке	по ссылке

Здесь «копия» означает вызов копирующего конструктора, «по ссылке» означает привязку параметра-ссылки к аргументу, «перем.» означает вызов *перемещающего конструктора*, если он доступен. Если же его нельзя вызвать, то будет выполнено копирование. Прочерк означает ошибку компиляции.

Перемещающий конструктор [*move constructor*] и **перемещающий оператор присваивания** [*move assignment operator*] принимают ссылку на временное значение своего типа и могут просто «забрать» его содержимое.

Например, в случае варианта нашего класса Matrix для способа 2 перемещающие конструктор и оператор присваивания можно реализовать так:

```
// Перемещающий конструктор.
Matrix(Matrix && other)
  : _rows(other._rows), _cols(other._cols),
    _data(other._data) {
  other._rows = other._cols = 0;
  other._data = nullptr;
}
// Перемещающий оператор присваивания.
Matrix & operator=(Matrix && other) {
  if (this != &other) {
    Matrix temp(move(other));
    this->swap(temp);
  }
  return *this;
}
```

Итак, мы «крадем» содержимое временного объекта, делая его пустым. После данной операции временный объект должен находиться в корректном состоянии, поэтому мы обнуляем его поля.

Внутри функции, принимающей параметр T && a, имя a имеет тип T&. Поэтому, чтобы вызвать именно перемещающий конструктор (при создании объекта temp), нужно явно привести тип other к Matrix&&. Это можно сделать с помощью стандартной функции move (определена в заголовочном файле **utility**).

Отметим также, что по умолчанию функции-члены могут быть вызваны хоть через обычную ссылку на объект, хоть через ссылку на временное значение. Если по каким-то причинам мы хотим различать эти варианты, то можно воспользоваться специальным синтаксисом объявления функций-членов:

```
class A {
  // ...
  void a();     // и для A&, и для A&&
  void b()&;    // только для A&
  void c()&&;   // только для A&&
  void d()&;    // вариант d() для A&
  void d()&&;   // вариант d() для A&&
```

Введение концепции перемещения влечет расширение правила трех до «правила пяти».

Правило пяти [*the Rule of Five*]. Если требуется определить что-то из следующего списка:

1. Деструктор.
2. Копирующий конструктор.
3. Копирующий оператор присваивания.
4. Перемещающий конструктор.
5. Перемещающий оператор присваивания.

То следует определить **все пять**.

Если не определена ни одна из этих пяти функций, то компилятор попробует их сгенерировать автоматически (все пять). Если мы определим какую-либо из первых трех, то компилятор

не будет генерировать перемещающие конструктор и оператор присваивания.

Если требуется явно указать компилятору, что мы не хотим какую-либо из функций, то ее можно объявить с директивой = **delete**:

```
class NonCopyable {
  // Запретим копирование.
  NonCopyable(NonCopyable const&) = delete;
  NonCopyable&
    operator=(NonCopyable const&) = delete;
};
```

Мы также можем попросить компилятор сгенерировать реализацию какой-либо из функций автоматически (если такой автоматический вариант определен по Стандарту и доступен для данного типа), объявив ее с директивой = **default**:

```
class StrangeThing {
  // Копирование определено нами.
  StrangeThing(StrangeThing const&);
  StrangeThing& operator=(StrangeThing const&);
  // Перемещение по умолчанию будет отсутствовать.
  // Попросим компилятор сделать его автоматически:
  StrangeThing(StrangeThing&&) = default;
  StrangeThing& operator=(StrangeThing&&) = default;
};
```

Стандартная библиотека предлагает класс, являющийся одной из простейших реализаций принципа управления ресурсом: unique_ptr (заголовочный файл **memory**). Объекты данного класса хранят указатели на другие объекты или массивы и удаляют их автоматически в деструкторе. Объекты unique_ptr нельзя копировать[103], но можно перемещать.

Создать объект unique_ptr можно с помощью стандартной функции make_unique. Следующий код создает в динамической

[103] Слово *unique* — «уникальный, единственный» в названии говорит о том, что ресурсом управляет ровно один объект.

памяти переменную **int**(10), которой будет управлять объект pn типа unique_ptr<**int**>:

```
auto pn = make_unique<int>(10);
```

Создать динамический массив можно, указав после типа [] и передав функции make_unique размер массива:

```
auto pa = make_unique<int[]>(10);
```

Вызов make_unique<T>(args) создает объект через **new** T(args), вызов make_unique<T[]>(n) создает массив через **new** T[n]{}.

Объект unique_ptr можно разыменовывать как указатель, но с ним нельзя выполнять арифметические действия. Если требуется получить собственно указатель, то для этого имеется функция-член get().

Класс Matrix на основе unique_ptr и способа 2 может быть построен следующим образом:

```
class Matrix {
  int _rows, _cols;
  unique_ptr<ET[]> _data;
public:
  // Конструктор по умолчанию (пустая матрица).
  Matrix(): _rows(0), _cols(0) {}
  // Конструктор матрицы заданных размеров.
  Matrix(int rows, int cols)
    : _rows(rows), _cols(cols),
      _data(make_unique<ET[]>(size_t(rows)*cols)) {}
  // Перемещающий конструктор.
  Matrix(Matrix && other)
    : _rows(other.rows()), _cols(other.cols()),
      _data(move(other._data)) {
    other._rows = other._cols = 0;
  }
  // Перемещающий оператор присваивания.
  Matrix & operator=(Matrix && other) {
    if (this != &other) {
      // Через перемещающий конструктор.
      Matrix temp(move(other));
      swap(temp);
```

```
    }
    return *this;
  }
  // Копирующий конструктор.
  Matrix(Matrix const & other)
    : Matrix(other.rows(), other.cols()) {
    // Скопировать элементы.
    auto const sz = size();
    auto const s = other.data();
    auto const d = data();
    for (size_t i = 0; i < sz; ++i)
      d[i] = s[i];
  }
  // Копирующий оператор присваивания.
  Matrix & operator=(Matrix const & other) {
    // Через конструктор копии и перемещающий =.
    return *this = Matrix(other);
  }

  // Проверка на пустоту.
  bool empty() const { return _data == nullptr; }
  // Сколько строк?
  int rows() const { return _rows; }
  // Сколько столбцов?
  int cols() const { return _cols; }
  // Обратиться к строке матрицы.
  ET * operator[](int row) {
    return data() + row*_cols;
  }
  ET const * operator[](int row) const {
    return data() + row*_cols;
  }
  // Сколько всего элементов в массиве?
  size_t size() const {
    return size_t(rows()) * cols();
  }

  // Прямой доступ к массиву.
  ET * data() { return _data.get(); }
  ET const * data() const { return _data.get(); }
```

```
  // Обменять содержимое двух матриц.
  void swap(Matrix & other) {
    std::swap(_rows, other._rows);
    std::swap(_cols, other._cols);
    _data.swap(other._data);
  }

  // ...
};
```

Применяя способ 1, можно построить массив указателей unique_ptr<ET[]>:

```
class Matrix {
public:
  // Конструктор по умолчанию.
  Matrix(): _rows(0), _cols(0) {};
  // Конструктор матрицы заданных размеров.
  Matrix(int rows, int cols);
  // Копирующий конструктор.
  Matrix(Matrix const &);
  // Копирующий оператор присваивания.
  Matrix & operator=(Matrix const &);
  // Перемещающий конструктор.
  Matrix(Matrix &&);
  // Перемещающий оператор присваивания.
  Matrix & operator=(Matrix &&);
  // Заполнить матрицу константой.
  void fill(ET val);
  // Обменять содержимое двух матриц.
  void swap(Matrix &);
  // Проверка на пустоту.
  bool empty() const { return _data == nullptr; }
  // Сколько строк?
  int rows() const { return _rows; }
  // Сколько столбцов?
  int cols() const { return _cols; }
  // Сколько всего элементов?
  size_t size() const
  { return size_t(rows()) * cols(); }
```

```
  // Доступ к строке по ее индексу.
  ET * operator[](int row) {
    return _data[row].get();
  }
  ET const * operator[](int row) const {
    return _data[row].get();
  }

private:
  // Воспользуемся std::unique_ptr.
  using Row = unique_ptr<ET[]>;
  using Rows = unique_ptr<Row[]>;
  // Данные объекта.
  int _rows, _cols;
  Rows _data;
};

// Определения функций-членов.
// Перемещающий конструктор.
Matrix::Matrix(Matrix && other)
  : _rows(other.rows()), _cols(other.cols()),
    _data(move(other._data)) {
  other._rows = other._cols = 0;
}

// Перемещающий оператор присваивания.
Matrix & Matrix::operator=(Matrix && other) {
  if (this != &other) {
    // Через перемещающий конструктор.
    Matrix temp(move(other));
    swap(temp);
  }
  return *this;
}

// Конструктор матрицы заданных размеров.
Matrix::Matrix(int rows, int cols)
  : _rows(rows), _cols(cols) {
  // Создать массив указателей.
```

```
    _data = make_unique<Row[]>(rows);
    // Создать строки.
    for (int row = 0; row < rows; ++row)
      _data[row] = make_unique<ET[]>(cols);
    // В случае ошибки память будет освобождена.
}
```

Задания для самопроверки

У Ответьте на следующие вопросы.

- Сколько места в памяти займет двумерный массив размера $n \times m$ из элементов типа T, реализованный с помощью способа 3?
- Каков жизненный цикл объекта, являющегося значением глобальной переменной?
- Как проверить, что две ссылки ссылаются на один и тот же объект?
- Зачем нужно ключевое слово **this**?
- Чем отличаются поля, определенные в **private**-секции, от полей, определенных в **public**-секции?
- В чем отличие семантики ключевых слов **struct** и **class**, используемых для определения типа?
- Может ли **static** функция-член содержать модификатор **const** после списка параметров? Почему?
- Пусть дано определение:

```
class A {
  A(A const&);
  A& operator=(A const&);
  // Другие члены класса (не конструкторы)...
};
```

Какой набор конструкторов доступен для класса A? Кто может копировать объекты класса A?

- Пусть дано определение:

```
class B {
  B(int);
  B(A const&) = delete;
  B& operator=(B const&) = delete;
  // Другие члены класса (не конструкторы)...
};
```

Какой набор конструкторов доступен для класса B?

- Чего позволяют избежать перемещающие конструктор и оператор присваивания?

- Что такое «правило пяти»?

- Пусть дано определение класса:

```
size_t const BUFSZ = size_t(1) << 20;
class C {
  char * _inbuf, * _outbuf;
public:
  // Конструктор, выделяет место для буферов.
  C(): _inbuf(nullptr), _outbuf(nullptr) {
    _inbuf = new char[BUFSZ];
    _outbuf = new char[BUFSZ];
  }
  // Деструктор, освобождает память.
  ~C() { delete[] _outbuf; delete[] _inbuf; }
  // ...
};
```

Какие недостатки оно имеет? Как их исправить?

- Зачем нужна стандартная функция move?

- Дан следующий код:

```
#include <iostream>
using namespace std;
struct A {
  void f() & { cout << "A"; }
  void f() && { cout << "B"; }
};

void f(A && a) { a.f(); }

int main() {
  A{}.f();
  f(A{});
  return 0;
}
```

  Какое сообщение он выведет?

У Пусть дан двумерный массив, упакованный в одномерный (способ 2). Напишите функцию, выполняющую его транспонирование на месте.

Глава 12

Введение в численные методы

12.1. Погрешность и точность

Поскольку множество $\mathbb{R}$ несчетно, то выполнять абсолютно точные вычисления в действительных числах на цифровом компьютере в общем случае невозможно. Даже если ограничиться операциями, заданными на счетном поле $\mathbb{Q}$, то можно попасть в ситуацию, когда представление используемых чисел неограниченно растет, что заодно приводит и к опережающему росту вычислительной сложности. Естественным выходом из этой ситуации является выполнение вычислений *приближенно*, используя числа, размер представления которых ограничен или вовсе постоянен. Последнее справедливо для всех встроенных числовых типов C++. Использование таких типов данных для выполнения вычислений приводит к появлению *погрешности вычислений* — отклонения от результата, который мог бы получиться при выполнении абсолютно точных вычислений. Однако вычисления ограниченной точности — не единственный источник погрешности. Погрешность результата вычислений может складываться из нескольких погрешностей:

- *погрешности модели*: рассматривая задачу, мы подменяем реальный мир его упрощенным описанием — моделью. Это упрощение влечет отклонение результатов рассуждений и вычислений от того, что происходит в реальности;

- *погрешности исходных данных*: если в качестве исходных данных для вычислений используются результаты измерений физических объектов, то они обязательно включают в себя погрешность измерения;

- *погрешности метода*: используемые методы вычисления (например, решения уравнения и т. п.), как правило, не дают точный результат за конечное число шагов (операций), даже если мы используем абсолютно точные представления чисел;

- *погрешности вычислений* (*накопление округлений*): сюда относят погрешность, связанную с ограниченным представлением чисел и соответственно ограниченной точностью выполнения над ними арифметических операций и вычисления элементарных функций.

Первые два вида погрешности можно объединить в общую *погрешность задачи*, которая является неустранимой с точки зрения вычислителя.

Пусть есть искомое значение x и некоторое $x^* \approx x$.

Абсолютная погрешность [*absolute error*] — мера *расстояния* между x и x^*:

$$\mathrm{err}_{abs} \triangleq \|x - x^*\|.$$

Если x и x^* — числа, то $\|x - x^*\| = |x - x^*|$ и имеет ту же физическую размерность, что и x. Если x и x^* — элементы $\mathbb{R}^n$, то в качестве расстояния обычно берут евклидову длину вектора разности.

Относительная погрешность [*relative error*] — безразмерная величина:

$$\mathrm{err}_{rel} \triangleq \frac{\mathrm{err}_{abs}}{\|x\|}.$$

Относительная погрешность не имеет смысла, если x может быть нулем. Поэтому для малых значений или значений, которые могут менять знак, обычно используется абсолютная погрешность. В прочих случаях относительная погрешность удобнее.

Предположим, что вычисления ведутся с абсолютной точностью. При сложении и вычитании величин абсолютная погрешность результата не превосходит суммы погрешностей операндов.

При умножении и делении ситуация несколько сложнее. Упрощая, можно сказать, что относительная погрешность результата в этом случае есть сумма относительных погрешностей операндов. Соответственно, при возведении в степень p (необязательно целую) имеем увеличение относительной погрешности в p раз (приближенно). Например:

$$x^* = (1 + \mathrm{err}_{rel})x, \quad (x^*)^2 = ((1 + \mathrm{err}_{rel})x)^2 = \\ = (1 + 2\mathrm{err}_{rel} + \mathrm{err}_{rel}^2)x^2 \approx (1 + 2\mathrm{err}_{rel})x^2.$$

Так как из-за округлений при вычислениях накапливается погрешность, вместо прямого сравнения чисел с плавающей точкой на равенство или неравенство обычно разумнее выполнять оценку расстояния между ними. В простейшем случае может использоваться сравнение по абсолютной величине погрешности. Этот вариант лучше всего подходит, когда надо проверять числа на равенство нулю:

```
inline bool are_close
    (double a, double b, double eps) {
  return fabs(a - b) <= eps;
}
```

Однако для сравнения произвольных чисел подобрать фиксированное значение константы eps часто не представляется возможным. В этом случае следует сравнивать числа по относительной величине разности. Здесь переменные a и b в некотором смысле равноценны, поэтому делим не на модуль a или

b, a на максимум из их модулей (функция fmax используется из соображения удобства, на деле она может оказаться довольно медленной, а игнорирование нечисел в сравнении не имеет практического смысла):

```
inline bool are_rel_close
    (double a, double b, double rel_eps) {
  return fabs(a - b) <=
    rel_eps * fmax(fabs(a), fabs(b));
}
```

При использовании форматов чисел с плавающей запятой точность выполнения операций может измеряться в единицах, равных весу младшего разряда множителя результата, называемых *ULP* (*units in the last place*). Например, расстояние между 1.0 и (1.0 + DBL_EPSILON) составляет 1 ULP. При удвоении числа вес ULP также удваивается, поэтому погрешность, выраженная в ULP, является относительной погрешностью.

При вычислении арифметических операций, квадратного корня и функции fma (слитое умножение-сложение) результат округляется к ближайшему представимому в текущем формате числу, т. е. имеет точность 0.5 ULP (при использовании режима округления к ближайшему представимому числу). Данная погрешность является лучшей теоретически возможной ввиду ограничений самого представления чисел. Одним из следствий такого округления является точный результат sqrt от квадрата целого числа (если он представим).

Сравнивать числа на близость «почти» в ULP (точнее «в эпсилонах») можно следующей простой функцией:

```
inline bool are_eps_close
    (double a, double b, double epsilons) {
  return fabs(a - b) <=
    (DBL_EPSILON*epsilons)*fmax(fabs(a), fabs(b));
}
```

Вычисление тригонометрических функций, степеней и логарифмов нередко выполняется не столь точно и может давать погрешность 1–2 ULP и более (зависит от реализации и кон-

кретной функции). Поэтому вычисление формул, содержащих такие функции, с одними и теми же значениями операндов может давать разные результаты на разных платформах.

Благодаря двоичному формату чисел IEEE-754, при интерпретации их как целых[104], модуль их разности в случае совпадения знака и порядка равен расстоянию между ними в ULP, но нужно учитывать тот факт, что бесконечности и нечисла будут находиться на конечном расстоянии от чисел (о функции memcpy см. на с. 289).

```
#include <cstring> // memcpy
#include <cstdint> // uint64_t
// Предположение: sizeof(double) == sizeof(uint64_t).
inline bool are_ulp_close
    (double a, double b, uint64_t ulps) {
  if (a == b) // случай +0, -0
    return true;
  // Извлечем двоичное представление.
  uint64_t a_bits, b_bits;
  memcpy(&a_bits, &a, sizeof(uint64_t));
  memcpy(&b_bits, &b, sizeof(uint64_t));
  // Если знак чисел разный, вернем false.
  if ((a_bits ^ b_bits) >> 63)
    return false;
  // Проверим собственно разность.
  return (a_bits - b_bits <= ulps)
       | (b_bits - a_bits <= ulps);
}
```

Наличие погрешности определяет точность результата. Понятие *точность* может быть истолковано двояко:

- *точность значения* [*accuracy*] — количество *верных цифр* в записи множителя значения;

- *точность представления* [*precision*] — количество значащих цифр в представлении множителя значения. На-

[104] На некоторых платформах может потребоваться изменение порядка байт.

пример, 24 двоичных цифры есть точность представления нормализованных чисел формата IEEE-754 Binary32.

Цифру числа называют *верной*, если его абсолютная погрешность не превосходит половины единицы соответствующего разряда. Таким образом, выполнение одной арифметической операции со значениями, представленными в плавающей точке точно, дает результат, все двоичные цифры которого верны (погрешность 0.5 ULP).

Естественная задача, называемая *обратной задачей теории погрешностей*[105], заключается в определении допустимого уровня погрешности задачи для получения результатов вычислений заданной точности с учетом других источников погрешности.

12.2. Некоторые частные вопросы

Функции, оперирующие представлением чисел

Стандартная библиотека включает ряд функций для манипуляций представлением чисел с плавающей запятой (некоторые из них уже встречались в главе 7):

- signbit(x) возвращает истину, если бит знака x установлен, иначе ложь;

- copysign(x, y) возвращает представление x, бит знака в котором заменен на бит знака из представления y;

- nextafter(x, y) возвращает следующее после x представимое число в направлении числа y. Если $x = y$, то возвращает y. Если любой из аргументов нечисло, возвращает нечисло;

- ldexp(x, n) вычисляет $x \cdot 2^n$. В случае переполнения получается бесконечность. Если $x = 0$ или $x = \pm\infty$, то функция вернет x;

[105] *Вержбицкий В. М.* Основы численных методов. 2-е изд. М., 2005. С. 16.

• scalb(x, n) добавляет n к экспоненте x. Данная функция эквивалентна ldexp на системах, где экспонента задает показатель степени двойки;

• frexp(x, p) (здесь p — указатель на **int**) выполняет разделение множителя и экспоненты. Если $x = \pm\infty$ или нечисло, то функция возвращает, соответственно, $\pm\infty$ или нечисло и записывает некое значение в $*$p. Если $x = 0$, то функция возвращает x и $*\text{p} = 0$. В иных случаях функция возвращает $y \in (-1, -\frac{1}{2}] \cup [\frac{1}{2}, 1)$ и записывает n в $*$p такое, что $x = y \cdot 2^n$;

• ilogb(x) возвращает целое число (экспоненту числа $x \neq 0$) $n = [\log_r |x|]$, где $[z]$ обозначает целую часть z, а r — основание степенного сомножителя числа с плавающей запятой (как правило, 2). Возвращает INT_MAX, если $x = \infty$ и особые константы FP_ILOGB0 для нуля, FP_ILOGBNAN для нечисел. Результат для чисел на 1 меньше того, что возвращает (по указателю) функция frexp;

• logb(x) аналогична ilog, но возвращает значение в плавающей точке. Бесконечность для $x = \pm\infty$ и $x = \pm 0$, нечисло для нечисел;

• modf(x, p) (здесь p — указатель на число с плавающей запятой) выполняет разделение целой и дробной части: $x = y + z$, $y \in \mathbb{Z}$, $z \in (-1, 1)$, знак z определяется знаком x. Функция записывает целую часть y по адресу p и возвращает дробную часть z. В случае, если $x = \pm\infty$, функция записывает x в $*$p и возвращает ± 0. В случае, если x — нечисло, функция записывает нечисло в $*$p и возвращает нечисло.

Используя функцию nextafter, можно, например, вывести все представимые неотрицательные числа типа **float** в консоль:

```
void print_all_nonnegative_floats_std() {
  for (float f = 0.f; f <= FLT_MAX;
       f = nextafter(f, INFINITY))
    cout << f << '\n';
}
```

То же самое можно сделать через обращение к представлению числа в плавающей точке как к целому (его увеличение на 1 дает в данном случае следующее представимое число, пока не будет достигнуто значение $+\infty$):

```
// Предположение: float == IEEE-754 Binary32.
void print_all_nonnegative_floats_mem() {
  for (uint32_t i = 0; i <= 0x7F800000u; i++) {
    float f;
    memcpy(&f, &i, sizeof(float));
    cout << f << '\n';
  }
}
```

Возведение в целую степень

Стандартная функция pow переводит целочисленные показатели в значения типа **double** и далее действует как базовая версия для нецелых показателей (хотя библиотеки могут реализовывать частные случаи отдельно, например, pow(x,2)).

Вычисление целочисленной степени прямым умножением не является хорошим способом реализации данной операции:

```
double bad_ipow(double x, unsigned n) {
  double p = 1.;
  while (n-- != 0) // постдекремент n
    p *= x;
  return p;
}
```

Количество умножений можно сделать логарифмическим по n, если воспользоваться ассоциативностью и коммутативностью произведения и двоичным представлением показателя степени:

$$\mathrm{n} = b_0 + 2b_1 + 4b_2 + \ldots + 2^{w-1}b_{w-1} = \sum_{i=0}^{w-1} 2^i \cdot b_i,$$

$$x^{\mathrm{n}} = x^{\sum_{i=0}^{w-1} 2^i \cdot b_i} = \prod_{i=0}^{w-1} x^{2^i \cdot b_i}.$$

Здесь w — количество бит в представлении n, b_i — значение i-го бита n (0 или 1).

Алгоритм можно записать так:

```
double ipow(double x, unsigned n) {
  double p = 1.;
  while (n != 0) {
    if (n & 1) // проверить младший бит
      p *= x;  // домножить на сомножитель
    n >>= 1;   // убрать младший бит
    x *= x;    // получить следующий сомножитель
  }
  return p;
}
```

О вычислении значений многочленов

Представим, что требуется вычислять значение функции

$$f(x) \triangleq x^4 - \frac{1}{2}x^3 + \frac{1}{2}x^2 - 2x + 1.$$

Наивный подход заключается в независимом вычислении всех степеней x, например, так:

```
float f_naive(float x) {
  return pow(x, 4) - pow(x, 3)/2
       + pow(x, 2)/2 - 2*x + 1; }
```

Можно попытаться улучшить эту реализацию, вынеся промежуточные степени в переменные:

```
float f_better(float x) {
  float const x2 = x*x;
  return x2*x2 - .5f*x2*x + 0.5f*x2 - 2.f*x + 1.f; }
```

Нетрудно заметить, что, по крайней мере, x2 можно вынести и сэкономить, таким образом, пару умножений:

```
float f_better2(float x) {
  float const x2 = x*x;
  return x2*(x2 - .5f*(x + 1.f)) - 2.f*x + 1.f; }
```

Необходимо помнить, что подобные преобразования, строго говоря, изменяют функцию, поскольку операции с числами, представленными в плавающей запятой, не являются ассоциативными (и умножение, и сложение).

Стандартным методом вычисления многочленов является *метод Горнера*[106], основанный на вынесении x и сведении вычисления многочлена к последовательности операций умножения и сложения:

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0 = \\ = ((\ldots(a_n x + a_{n-1})x + \ldots + a_2)x + a_1)x + a_0.$$

Данный метод можно применить и к произвольному многочлену, заданному массивом коэффициентов, и к конкретному многочлену:

```
float f_horner(float x) {
  return 1.f + x*(2.f + x*(0.5f + x*(0.5f + x))); }
```

При использовании целевой платформы, поддерживающей команды FMA, и соответствующих настроек компилятора[107] данный код может быть скомпилирован (не считая команд перемещения данных) в последовательность из четырех арифметических команд: сложения и трех слитых умножений-сложений (FMA), что можно представить в виде следующего псевдокода:

```
input r0;                  // r0 = x
r1 = 1.f;
r2 = 2.f;
r3 = 0.5f;
r4 = r0 + r3;              // r4 = x + 0.5f
r4 = fma(r4, r0, r3);      // r4 = r4*x + 0.5f
r4 = fma(r4, r0, r2);      // r4 = r4*x + 2.f
r0 = fma(r0, r4, r1);      // r0 = x*r4 + 1.f
return r0
```

[106] Также *схема Горнера*, по фамилии англ. математика У. Дж. Горнера (W. G. Horner, 1786–1837).

[107] В случае GCC это ключ `-mfma`.

Необходимо отметить, что результат FMA может отличаться от результата последовательности умножения и сложения (обычно только в последнем бите множителя, но иногда и кардинально: например, если при умножении произойдет переполнение).

Если мы хотим потребовать выполнения FMA, то можно задействовать стандартную функцию fma (из **cmath**):

```
float f_horner_fma(float x) {
  return fmaf(
           fmaf(
             fmaf(x + 0.5f,
             x, 0.5f),
           x, 2.f),
         x, 1.f); }
```

В этом случае слитое умножение-сложение будет выполняться даже на целевых платформах, аппаратно его не поддерживающих.

У Напишите функцию, принимающую число x и коэффициенты многочлена как массив чисел и вычисляющую значение этого многочлена в точке x методом Горнера.

Операцию сложение-умножение даже применяли как единицу вычислительной сложности («1 горнер»). Это связано с широким использованием многочленов для приближенного представления других функций. Более популярным является оценка ее как двух операций с плавающей запятой («2 FLOPS»). Например, при вычислении пиковой производительности процессора: пусть дан процессор, работающий на частоте 3 ГГц, состоящий из 8 «ядер», каждое из которых содержит два устройства, способных выполнять операции с плавающей запятой над векторами из 4 чисел Binary64 или 8 чисел Binary32 (ширина 256 бит), включая операцию FMA в конвейерном режиме (по одной за такт на каждом устройстве при отсутствии зависимостей). Тогда пиковая производительность данного процессора для Binary32 составит $3 \cdot 10^9$ (Гц) $\cdot 8 \cdot 2 \cdot 8 =$ $= 384 \cdot 10^9$ операций FMA в секунду или 768 GFLOPS.

Попробуем расставить скобки иначе:

$$x^4 - \frac{1}{2}x^3 + \frac{1}{2}x^2 - 2x + 1 =$$
$$= \left((2x+1) + \left(\frac{1}{2}x + \frac{1}{2}\right)x^2\right) + x^4.$$

Соответствующий код на C++

```
float f_estrin(float x) {
  auto const
    p1 = 2.f*x + 1.f,
    p2 = 0.5f*x + 0.5f,
    x2 = x*x;
  return p1 + (p2 + x2)*x2;
}
```

компилируется в иную последовательность команд, которую условно можно представить следующим образом:

```
input r0;                    // r0 = x
r1 = 1.f;
r2 = 2.f;
r3 = 0.5f;
r4 = r0 * r0;                // r4 = x*x
r3 = fma(r3, r0, r3); // r3 = 0.5f*x + 0.5f
r2 = fma(r2, r0, r1); // r2 = 2.f*x + 1.f
r3 = r3 + r4;
r0 = fma(r3, r4, r2); // r0 = r3*r4 + r2
return r0
```

Представим, что у нас «сдвоенное» вычислительное устройство, которое способно выполнять две независимых команды одновременно. Тогда код можно переписать так:

```
input r0;
r1 = 1.f;    r2 = 2.f;   r3 = 0.5f;
r4 = r0 * r0             | r3 = fma(r3, r0, r3);
r2 = fma(r2, r0, r1)     | r3 = r3 + r4;
r0 = fma(r3, r4, r2);
return r0
```

Теперь у нас три вычислительных шага, а не четыре, как в методе Горнера. Поэтому на современных процессорах, как правило, способных выполнять одновременно по две разных операции с числами с плавающей запятой (два конвейера FPU), такой код может оказаться быстрее даже несмотря на тот факт, что операций выполняется больше.

Основная идея подхода (известного также как *метод Эстрина*[108]) — разбивать вычисление на пары независимых двучленов. Наибольшую эффективность он имеет при вычислении многочленов степени $2^n - 1$, $n \in \mathbb{N}$. Например:

$$a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 =$$
$$= ((a_7x + a_6)x^2 + (a_5x + a_4))x^4 + (a_3x + a_2)x^2 + (a_1x + a_0).$$

Здесь четыре двучлена нижнего уровня и x^2 могут быть вычислены одновременно. Затем одновременно можно вычислить два двучлена следующего уровня и x^4. После чего одно умножение-сложение даст итоговый результат.

Интерполяция

Интерполяция[109] **функции** — подмена функции, с известным конечным набором значений (узлов), заданной на отрезке функцией, которая в узлах принимает те же значения, что и исходная.

Интерполяция является одним из важнейших элементов вычислений, позволяя подменять сложные или неизвестные функции известными и (сравнительно) легко вычислимыми функциями. Здесь мы не будем рассматривать общие аспекты интерполяции и, в частности, вопрос ее точности. Вместо этого приведем некоторые примеры.

---

[108] См.: *Green R.* Faster Math Functions // SlideServe : [site]. URL: `https://www.slideserve.com/kenley/faster-math-functions` (дата обращения: 29.07.2020).

[109] От лат. *interpolo* — «подновлять, подкрашивать, подделывать», *inter-* — «между» и *polio* — «разглаживать, полировать».

Предположим, что вся информация о функции $f(x)$ исчерпывается одним ее известным значением f_0 в одной точке (узле) x_0:

$$f(x_0) = f_0.$$

Как моделировать такую функцию? Простейший вариант заключается в том, чтобы положить ее константой:

$$f(x) \approx F(x), \quad F(x) \equiv f_0, \quad x \in \mathbb{R}.$$

Однако вариант задачи с одним узлом не слишком интересен с практической точки зрения. Совсем другое дело, когда известно два узла:

$$f(x_0) = f_0, \quad f(x_1) = f_1.$$

Простейшим решением будет провести через них прямую:

$$F(x) = \frac{f_0(x_1 - x) - f_1(x_0 - x)}{x_1 - x_0} = \frac{(f_0 x_1 - f_1 x_0) + (f_1 - f_0)x}{x_1 - x_0}.$$

Такая интерполяция называется *линейной*. Ее практический смысл следует из того факта, что дифференцируемые функции на малых отрезках ведут себя подобно линейным:

$$f(x + h) = f(x) + f'(x)h + o(h).$$

Если у нас задано три узла: (x_0, f_0), (x_1, f_1), (x_2, f_2), то можно использовать *параболическую интерполяцию*:

$$F(x) = \frac{(x_1 - x)(x_2 - x)}{(x_1 - x_0)(x_2 - x_0)} f_0 + \frac{(x_0 - x)(x_2 - x)}{(x_0 - x_1)(x_2 - x_1)} f_1 + \\ + \frac{(x_0 - x)(x_1 - x)}{(x_0 - x_2)(x_1 - x_2)} f_2.$$

Далее, по четырем узлам можно построить *кубическую интерполяцию*, но отдельную формулу мы здесь приводить не будем. Представленную конструкцию можно обобщить: многочлен степени n способен интерполировать функцию, заданную

$n+1$ узлом. Приведенные выше формулы для линейной и параболической интерполяции являются частными случаями интерполяционного многочлена в форме Лагранжа[110]:

$$F(x) = \sum_{i=0}^{n} \left(f_i \prod_{j=0, j\neq i}^{n} \frac{x_j - x}{x_j - x_i} \right).$$

Интересный вариант кубической интерполяции можно получить, если заданы два узла, в которых мы знаем не только значение интерполируемой функции, но и угол касательной: (x_0, f_0, α_0), (x_1, f_1, α_1). Пусть $f'_i = \operatorname{tg}\alpha_i$ $(i = 0, 1)$ — значение производной в узле. Тогда кубическая интерполяция будет иметь следующий вид:

$$F(x) = (\tau f_1 - \sigma f_0) + \tau\sigma((1-2\tau)(f_1 - f_0) + \\ + (x_1 - x_0)(\sigma f'_0 + \tau f'_1)),$$

где

$$\tau = \frac{x - x_0}{x_1 - x_0}, \quad \sigma = \tau - 1 = \frac{x - x_1}{x_1 - x_0}.$$

У Реализуйте класс Cubic, конструктор которого принимает параметры x_i, f_i, α_i типа **double** и сохраняет необходимые промежуточные константы в скрытые поля объекта. Пусть класс предоставляет функцию-член

double operator()(**double** x) **const**;

вычисляющую $F(x)$ в соответствии с указанной выше формулой кубической интерполяции.

Компенсационное суммирование

Прямое суммирование массива значений с плавающей запятой может давать большое накопление погрешности, особенно, если суммируются числа разного знака и сильно различающиеся по величине. Но даже в случае суммирования одного

[110] По фамилии предложившего такую форму фр. математика и механика Ж. Л. Лагранжа (J. L. Lagrange, 1736–1813).

и того же значения возможно превышение 100% относительной погрешности. В качестве простого примера приведем код, суммирующий миллиард единиц:

```
float sum = 0;
for (long i = 0; i < 1000'000'000; ++i)
  sum += 1.f;
cout << sum; // ???
```

Обратите внимание, что никакой перестановкой слагаемых результат не может быть улучшен, поскольку здесь все слагаемые равны друг другу.

Без учета возможного переполнения прямое суммирование чисел одного знака дает относительную погрешность $O(n)\varepsilon$, где n — число слагаемых, ε — вес младшего бита множителя.

У. М. Кэхэном[111] был предложен метод, получивший известность под названием *компенсационное суммирование*. Его идея заключается в вычислении *компенсации* каждой операции суммирования, как разности вычисленной суммы и слагаемых. Из-за промежуточных округлений это может быть не нуль. Компенсацию можно использовать для поправки суммы:

```
float sum_kahan(float const x[], size_t n) {
  float sum = 0.f, comp = 0.f;
  for (size_t i = 0; i < n; ++i) {
    // В случае включенной агрессивной оптимизации
    // операций с плавающей точкой компилятор может
    // сократить этот код до простого суммирования.
    float const y = x[i] - comp, t = sum + y;
    comp = (t - sum) - y;
    sum = t;
  }
  return sum;
}
```

[111] См.: *Kahan W.* Further remarks on reducing truncation errors // Comm. ACM. 1965. № 8(1):40. P. 40–48.

Существует оценка абсолютной погрешности суммирования данным методом[112]:

$$\left(2\varepsilon + O(n)\varepsilon^2\right) \frac{\sum_{i=0}^{n-1} |x_i|}{|\sum_{i=0}^{n-1} x_i|}.$$

Несколько улучшенная версия предложена Ноймайером[113]:

```
float sum_neumaier(float const x[], size_t n) {
  float sum = 0.f, comp = 0.f;
  for (size_t i = 0; i < n; ++i) {
    float const xi = x[i], t = sum + xi;
    if (fabsf(sum) >= fabsf(xi))
      comp += (sum - t) + xi;
    else
      comp += (xi - t) + sum;
    sum = t;
  }
  return sum + comp;
}
```

Произведение без промежуточных переполнений

В процессе вычисления произведения элементов последовательности чисел с плавающей запятой может оказаться, что диапазона экспоненты не хватает для представления величины промежуточного результата, в то время как правильно вычисленный конечный результат представи́м.

Например, предста́вим последовательность, состоящую из двух тысяч двоек и двух тысяч «половинок» (0.5). Произведение равно единице. Однако даже при использовании формата Binary64 еще на первой половине произойдет переполнение, так

---

[112] См.: *Higham N.* The accuracy of floating point summation // SIAM J. Sci. Comput. 1993. № 14(4). P. 783–799.

[113] См.: *Neumaier A.* Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen // Zeitschrift fur Angewandte Math. und Mech. 1974. № 54(1). S. 39–51.

как 2^{2000} слишком велико, и общий результат последовательного перемножения получится равным бесконечности. Если перемножать с конца к началу, то произойдет исчезновение, так как 2^{-2000} слишком мало и общий результат получится равным нулю.

Бороться с переполнением можно по-разному. Например, можно воспользоваться формулой:

$$\prod_{i=1}^{n} |x_i| = \exp \sum_{i=1}^{n} \ln |x_i|.$$

Однако обычно логарифмирование вычислительно дорого, и само преобразование может резко увеличить погрешность.

Другой способ заключается в отделении порядка от множителя. Для этого можно использовать функции ldexp и frexp, но на практике они могут иметь довольно низкое быстродействие. Поэтому ниже читателю предлагается алгоритм, основанный на домножении на фиксированные степени двойки, дающие добавление целого числа к порядку и не влияющие на биты множителя результата.

Итак, обратим внимание на тот факт, что нормальные числа в формате Binary32 (в качестве примера), находятся в полуинтервале $[2^{-126}, 2^{128})$, поэтому перемножение любой пары чисел из отрезка $[2^{-63}, 2^{63}]$ даст нормальное число. Это означает, что если текущее произведение и сомножители «загонять» в отрезок $[2^{-63}, 2^{63}]$ (домножая на 2^{-63} или на 2^{63} в случае нормальных чисел), то переполнения или исчезновения не случится. При этом необходимо запоминать множитель — соответствующую степень двойки.

```
float cproduct(float const a[], size_t n) {
  float p = 1.f; // текущее произведение
  int e = 0;     // сколько раз p домножить на 2^63
  int s = 0;     // знак произведения
  // Инвариант цикла:
  // p принадлежит [0x1p-63, 0x1p+63].
  for (size_t i = 0; i < n; ++i) {
```

```
    float x = a[i];     // следующий сомножитель
    // Поменять знак?
    if (signbit(x)) {
      x = -x;
      s = 1 - s;
    }
    // Субнормальное или нуль?
    if (x < 0x1p-126f) {
      if (x == 0) // произведение обнулилось?
        return s? -0.f: +0.f;
      e -= 2;
      x *= 0x1p+126f;
    }
    else if (x < 0x1p-63f) { // маленькое?
      --e;
      x *= 0x1p+63f;
    }
    else if (0x1p+63f < x) { // большое?
      if (isinf(x)) // произведение бесконечно?
        return s? -INFINITY: +INFINITY;
      ++e;
      x *= 0x1p-63f;
    }

    p *= x; // домножить
    // Обеспечить выполнение инварианта цикла.
    if (p < 0x1p-63f) { // маленькое?
      --e;
      p *= 0x1p+63f;
    }
    else if (0x1p+63f < p) { // большое?
      ++e;
      p *= 0x1p-63f;
    }
  }
  // Результат не может быть представим в случае
  // e < -3 или e > 3.
  if (e < -3)
    return s? -0.f: +0.f;
  if (3 < e)
```

```
    return s? -INFINITY: +INFINITY;
  // Домножим на нужное количество 2^-63.
  if (e < 0) {
    do p *= 0x1p-63f; while (++e != 0);
  }
  else if (0 < e) {
    do p *= 0x1p+63f; while (--e != 0);
  }
  // Результат.
  return s? -p: p;
}
```

Вычисление длины вектора

Линейное пространство, дополненное определением *нормы*, называется **нормированным пространством** [*normed space*]. **Норма** [*norm*] в пространстве V над полем K — отображение $\|\cdot\|\colon V \mapsto \mathbb{R}$, удовлетворяющее следующим свойствам (*аксиомы нормы*):

1. Если $\|x\| = 0$, то x есть нулевой вектор.

2. Неравенство треугольника:

$$(\forall x \in V)(\forall y \in V)\ \|x + y\| \leqslant \|x\| + \|y\|.$$

3. Вынесение модуля скалярного множителя:

$$(\forall a \in K)(\forall x \in V)\ \|ax\| = |a|\|x\|.$$

Здесь полагается, что в K задана операция модуля $|\cdot|\colon K \mapsto \mathbb{R}$. Модуль можно считать частным случаем нормы (для одномерного пространства).

Норму можно использовать для определения расстояния между векторами (*метрики*): $\rho(x, y) = \|x - y\|$. И наоборот, если задана метрика, то норму можно определить через расстояние до нулевого вектора: $\|x\| = \rho(x, \mathbf{0})$.

Если задано скалярное произведение векторов $x \cdot y$, то можно определить *естественную норму* $\|x\| = \sqrt{x \cdot x}$.

И метрика и скалярное произведение имеют ряд собственных свойств (аксиомы метрики, аксиомы скалярного произведения), которые мы здесь приводить не будем.

Популярным семейством норм на пространствах $\mathbb{R}^n$ и $\mathbb{C}^n$ являются *гёльдеровы нормы* ($p \geqslant 1$):

$$\|x\|_p = \sqrt[p]{\sum_{i=1}^{n} |x_i|^p}, \ p \geqslant 1.$$

Частные случаи:

- норма-1 $\|x\|_1 = \sum_{i=1}^{n} |x_i|$ — сумма модулей координат;
- норма-максимум $\|x\|_\infty = \max_{1 \leqslant i \leqslant n} |x_i|$ — максимум из модулей координат;
- евклидова норма $\|x\|_2 = \sqrt{\sum_{i=1}^{n} |x|^2}$.

Заметим, что с точки зрения вычисления данные частные случаи весьма различаются между собой: норму-максимум легко вычислить *точно*; для уменьшения погрешности нормы-1 ее можно вычислить с помощью компенсационного суммирования; а вот наиболее известная евклидова норма на практике может вызвать ряд проблем. Во-первых, это возможное переполнение или исчезновение при возведении в квадрат, что может привести к принципиальному падению точности. Во-вторых, это накопление погрешности при промежуточных операциях. Для решения этих проблем в случае пространств $\mathbb{R}^2$, $\mathbb{R}^3$ стандарт C++ предлагает функцию hypot (**cmath**).

Данная функция гарантирует отсутствие промежуточного переполнения и исчезновения. Как правило, она также дает погрешность менее 1 ULP.

Способ, основанный на вынесении максимума:

$$\sqrt{x^2 + y^2} = \max(|x|, |y|) \sqrt{1 + \frac{\min(|x|, |y|)}{\max(|x|, |y|)}}$$

может быть выражен в виде следующего кода:

```
float hypot_v1(float x, float y) {
  x = fabsf(x);
  y = fabsf(y);
  auto const
    min_ = x < y? x: y,
    max_ = x < y? y: x;
  if (max_ == 0.f || isinf(max_))
    return max_;
  auto const
    q = min_ / max_;
  return max_*sqrtf(1.f + q*q);
}
```

Данный код позволяет избежать ошибочного переполнения и исчезновения, но имеет в среднем погрешность даже хуже, чем прямое вычисление sqrtf(x*x + y*y), не говоря уже о скорости его работы.

Практичнее применить тот же прием, что был применен при реализации функции cproduct: вынесение степени двойки. Заметим также, что требуется определенное внимание к субнормальным аргументам:

Пример 12.1. Гипотенуза: отделение степени двойки

```
float hypot_v2(float x, float y) {
  x = fabsf(x);
  y = fabsf(y);
  float
    min_ = x < y? x: y,
    max_ = x < y? y: x,
    mul, invmul;
  if (max_ <= 0x1p-126f) {
    invmul = 0x1p-126f; // 0x1p-23 <= max_ <= 1
    mul = 0x1p+126f;    // 0x1p-23 <= min_
  }
  else if (max_ <= 0x1p-63f) {
    invmul = 0x1p-87f; // 0x1p-39 <= max_ <= 0x1p+24
    mul = 0x1p+87f;    // 0x1p-63 <= min_
  }
```

```
  else if (max_ <= 1.f) {
    invmul = 0x1p-24f; // 0x1p-39 <= max_ <= 0x1p+24
    mul = 0x1p+24f;    // 0x1p-63 <= min_
  }
  else if (max_ <= 0x1p+63f) {
    invmul = 0x1p+39f; // 0x1p-39 <= max_ <= 0x1p+24
    mul = 0x1p-39f;    // 0x1p-63 <= min_
  }
  else {
    invmul = 0x1p+102f; // 0x1p-38 <= max_ < 0x1p+26
    mul = 0x1p-102f;    // 0x1p-63 <= min_
  }

  max_ *= mul;
  min_ *= mul;
  if (max_ + min_ == max_)
    return invmul * max_;
  return invmul * sqrtf(max_*max_ + min_*min_);
}
```

Данная функция приведена для случая **float**. Однако когда доступна быстрая аппаратная реализация вдвое более широкой арифметики, использовать все такие приемы неразумно, поскольку элементарное:

```
float hypot_v3(float x, float y) {
  double const u = x, v = y;
  return float(sqrt(u*u + v*v));
}
```

гарантирует нам отсутствие ошибочных переполнений и исчезновений, дает точность почти 0.5 ULP и, вдобавок, работает быстрее предыдущих версий.

12.3. Решение уравнений

Пусть дано квадратное уравнение:

$$ax^2 + bx + c = 0.$$

Требуется найти его корни.

Предположим, что дискриминант

$$D = b^2 - 4ac$$

неотрицателен. Тогда, вспомнив школьный материал, можно записать:

$$x_{1,2} = \frac{-b \pm \sqrt{D}}{2a}.$$

Что здесь может быть не так?

Представьте ситуацию, когда $\frac{ac}{b} \approx 0$. В этом случае в одном из корней (с «+») получаем вычитание близких значений и резкий рост относительной погрешности из-за потери верных цифр. Этого можно избежать, если использовать для вычисления корней другую формулу[114]:

$$q = -\frac{1}{2}\left(b + \operatorname{sgn}(b)\sqrt{D}\right),$$
$$x_1 = \frac{q}{a}, \quad x_2 = \frac{c}{q}.$$

Теперь рассмотрим общую задачу. Пусть даны абсолютная погрешность err, непрерывная функция $f\colon \mathbb{R} \mapsto \mathbb{R}$ и $a < b$ такие, что $f(a)f(b) < 0$. Требуется найти $[a', b'] \subseteq [a, b]$ такой, что $b' - a' \leqslant \text{err}$ и

$$(\exists x_0 \in [a', b'])\ f(x_0) = 0.$$

У Как показать, что данная задача всегда имеет решение (при отсутствии вычислительной погрешности)?

Таким образом, искомые a', b' задают достаточно маленький интервал, на котором есть корень уравнения $f(x) = 0$. Если в качестве ответа требуется одно число, то можно взять $\frac{1}{2}(a' + b')$.

[114] См.: *Press W. H., Teukolsky S. A., Vetterling W. T., Flannery B. P.* Numerical Recipes: The Art of Scientific Computing. 3rd ed. Cambridge, 2007. P. 227.

Значение $|f(x_*)|$ для любого x_*, считающегося приближенно равным корню уравнения, называется *невязкой* [*residual*]. Необходимое ограничение сверху на невязку численного решения называется *допуском* [*tolerance*]. В задаче, поставленной выше, допуск не фигурирует, хотя технически данным термином могут обозначать любой критерий точности, например, в нашем случае абсолютную погрешность по решению err.

Простейшим и одновременно самым надежным и предсказуемым методом решения данной задачи является *метод бисекции*[115], заключющийся в повторяющемся разбиении текущего отрезка $[a, b]$ пополам и вычислением знака $f(m)$ в средней точке $m = 0.5(a + b)$. Если мы не попали в корень, то знак в этой точке должен не совпадать либо со знаком $f(a)$, либо со знаком $f(b)$, а значит, либо на участке $[a, m]$, либо на участке $[m, b]$ гарантированно[116] есть хотя бы один корень. Заменим, соответственно, либо b на m, либо a на m и повторим все сначала.

Как видно, в процессе мы либо попадаем однажды в корень, либо имеем последовательность отрезков $\{[a_i, b_i]\}$ такую, что:

1. Каждый следующий отрезок лежит в предыдущем: $[a_{i+1}, b_{i+1}] \subset [a_i, b_i]$.

2. Каждый следующий отрезок вдвое короче предыдущего: $b_i - a_i = 2(b_{i+1} - a_{i+1})$.

3. Значения функции на концах каждого из отрезков имеют разный знак: $f(a_i)f(b_i) < 0$.

Благодаря (1) $a_i < b_j$ для любых i и j.

Благодаря (2) отрезки стягиваются в точку: $b_i - a_i = (b_0 - a_0)2^{-i}$ (можно взять $a_0 = a$, $b_0 = b$ — начальный отрезок).

[115] От лат. *bi-* — «дву-» и *sectio* — «рассекание, разделение».

[116] Для доказательства этого факта можно использовать сам метод бисекции!

Можно даже сразу указать необходимое число шагов для достижения заданной точности результата: $n = \lceil \log_2 \frac{b-a}{\mathrm{err}} \rceil$. Итак, $\lim\limits_{i\to\infty} a_i = \lim\limits_{i\to\infty} b_i = x_*$, причем $a_i < x_* < b_i$.

Благодаря (3) функция с одного конца всегда меньше нуля, а с другого конца всегда больше нуля. Но поскольку функция непрерывна, то $f(x_*) = 0$.

```
// Где-то определена функция f(x).
float f(float);
// Ищем корень на [a, b] с точностью err.
float bisect(float & a, float & b, float err) {
  auto const m = (a + b)/2;
  if (b - a <= err)
    return m;
  return signbit(f(a)) != signbit(f(m))?
    bisect(a, b = m, err) : bisect(a = m, b, err);
}
```

Данная реализация имеет ряд недостатков. Для начала избавимся от рекурсии, заменив рекурсивный вызов на цикл:

```
float bisect(float & a, float & b, float err) {
  while (true) {
    auto const m = (a + b)/2;
    if (b - a <= err)
      return m;
    if (signbit(f(a)) != signbit(f(m)))
      b = m;
    else
      a = m;
  }
}
```

Теперь должно быть очевидно, что мы на каждой итерации вычисляем одно значение функции повторно — а именно f(a). Избавимся от этого:

```
float bisect(float & a, float & b, float err) {
  for (float fa = f(a);;) {
    auto const m = (a + b)/2;
    if (b - a <= err)
```

```
      return m;
    auto const fm = f(m);
    if (signbit(fa) != signbit(fm))
      b = m;
    else // заменить a и fa
      a = m, fa = fm;
  }
}
```

Наконец, можно попасть в неприятную ситуацию, когда из-за ограниченной точности представления чисел окажется, что m округлится до a или до b. В такой ситуации продолжать бессмысленно.

```
float bisect(float & a, float & b, float err) {
  for (float fa = f(a);;) {
    auto const m = (a + b)/2;
    if (b - a <= err || a == m || b == m)
      return m;
    auto const fm = f(m);
    if (signbit(fa) != signbit(fm))
      b = m;
    else // заменить a и fa
      a = m, fa = fm;
  }
}
```

Неудобной особенностью текущей реализации является также тот факт, что функция f фактически «зашита» в программу, ее нельзя сменить во время работы, разве что выбор конкретных вычислений будет помещен в саму f. C++ позволяет передавать функции по ссылке или указателю (аналогично в C), чем мы и воспользуемся.

Система типов C++ включает в себя формальные функциональные типы, которые, однако, нельзя использовать для определения переменных. Но можно определять переменные, имеющие тип указателя или ссылки на «значение» функционального типа (множество значений состоит из всех функций с теми же сигнатурой и типом возвращаемого значения).

Их описание напоминает типы массивов. Пусть R, A и B — типы. Дано объявление функции:

```
R f(A, B); /* произвольный список аргументов,
 в том числе пустой — () */
```

Тогда:

- R — тип возвращаемого значения;
- R(A, B) — формальный функциональный тип, которому принадлежит f и все прочие функции, принимающие пару значений типов A и B и возвращающие R;
- R(*)(A, B) — тип указателя на функцию f, т. е. &f;
- R(&)(A, B) — тип ссылки на функцию f.

Имя функции автоматически приводится к ссылке или указателю на эту функцию. Функцию по указателю или ссылке можно вызвать точно так же, как по «настоящему» имени, т. е. поставив после имени переменной скобки и перечислив в них аргументы вызова.

Добавим f в качестве параметра-ссылки:

```
float bisect(
    float (&f)(float),
    float & a, float & b, float err) {
  for (float fa = f(a);;) {
    auto const m = (a + b)/2;
    if (b - a <= err || a == m || b == m)
      return m;
    auto const fm = f(m);
    if (signbit(fa) != signbit(fm))
      b = m;
    else // заменить a и fa
      a = m, fa = fm;
  }
}
```

12.4. Поиск экстремума

Функция **унимодальна** на отрезке $[a, b]$, если существует $m \in [a, b]$ такое, что функция строго убывает на $[a, m]$ и строго возрастает на $[m, b]$. Таким образом, унимодальная функция достигает единственного на $[a, b]$ минимума в точке m.

Минимум унимодальной функции можно найти с заданной точностью методом, известным как *троичный поиск* [*ternary search*].

Его идея заключается в следующем. Пусть известны значения минимизируемой функции $f(x)$ в точках a и b, а также в двух точках c и d на $[a, b]$: $a < c < d < b$. Теперь возможны три ситуации:

1. $f(c) = f(d)$, минимум лежит на $[c, d]$.
2. $f(c) < f(d)$, минимум лежит на $[a, d]$.
3. $f(c) > f(d)$, минимум лежит на $[c, b]$.

Далее действуем аналогично методу бисекции.

Частный случай троичного поиска — *метод золотого сечения* [*golden ratio method*]. Данный метод определяет *как* выбирать средние точки на $[a, b]$ в постоянной пропорции таким образом, чтобы сходимость была наиболее быстрой в худшем случае.

Этого можно добиться, если делить отрезок в пропорции $1 : \phi$, где ϕ — «золотое сечение». Для любых $0 < y < x$ таких, что $\phi = \frac{x}{y}$, выполняется также $\frac{x+y}{x} = \frac{x}{y} = \phi$. Отсюда легко получить тождество $\phi = 1 + \phi^{-1}$, позволяющее вычислить ϕ как положительный корень квадратного уравнения.

Итак, алгоритм можно записать следующим образом:

1. Положить $c = b - (b - a)/\phi$.
2. Положить $d = a + (b - a)/\phi$.
3. Если $((d - c) \leqslant \mathrm{err}) \vee (c \leqslant a) \vee (b \leqslant d)$, то закончить.

4. Если $f(c) < f(d)$, то положить:
   - $b = d$;
   - $d = c$;
   - $c = b - (b - a)/\phi$.
5. Иначе если $f(d) < f(c)$, то положить:
   - $a = c$;
   - $c = d$;
   - $d = a + (b - a)/\phi$.
6. Иначе положить:
   - $a = c$;
   - $b = d$;
   - $c = b - (b - a)/\phi$;
   - $d = a + (b - a)/\phi$.
7. Повторить с п. 3.

У Реализуйте метод золотого сечения для поиска минимума функции, передаваемой по ссылке.

Задания для самопроверки

У Ответьте на следующие вопросы.

- Какая погрешность неустранима с точки зрения вычислителя?
- Если относительная погрешность a равна ε, то чему (приблизительно) равна относительная погрешность $\sqrt{|a|}$?
- Сколько верных цифр в записи числа 1.33567, если его абсолютная погрешность имеет величину порядка 10^{-4}?

- Как проверить, что два числа имеют разный знак?
- Какая операция является основной в методе Горнера?
- Какова пиковая производительность (FLOPS, Binary32) графического процессора, имеющего 4 096 параллельно работающих вычислительных устройств, способных каждое за один такт выполнять одну операцию FMA, если этот процессор работает на тактовой частоте 1 750 МГц?
- Какую задачу решает компенсационное суммирование?
- Зачем нужны стандартные функции log1p и exp1m?
- Во сколько раз укорачивается отрезок поиска на каждой итерации метода золотого сечения?

У Напишите функцию, которая проверяет близость двух значений по абсолютной и относительной величине (связка *или*).

У Реализуйте версию функции cproduct (произведение элементов массива) для чисел типа **double** в предположении, что этот тип использует представление IEEE-754 Binary64.

У Реализуйте версию функции hypot_v2 (пример 12.1) для чисел типа **double** в предположении, что этот тип использует представление IEEE-754 Binary64.

Глава 13

Си-строки

Несмотря на то, что стандартная библиотека C++ предлагает более удобный строковый тип string, строки «в стиле языка C» играют весьма значимую роль.

Во-первых, существует множество библиотек либо на языке C, либо использующих иные строковые типы или представления строк, и си-строки выступают как общедоступный базовый формат представления текста.

Во-вторых, уровень языка C является своего рода «общим знаменателем» между разными языками программирования и даже разными компиляторами C++, позволяя организовать взаимодействие между ними.

Наконец, при работе на ограниченных платформах может быть недоступен даже тип string (например, из-за отсутствия динамической памяти).

Таким образом, программист на C++ должен уметь работать с си-строками.

Си-строка (C-строка) или **нуль-терминированная строка** [*zero-terminated string*] — структура данных, представляющая собой массив символов, последний из которых (и только он) является нулевым.

Таким образом, строкой можно оперировать только по указателю на начальный символ, не храня ее размер явно. Так как

нулевой символ маркирует конец си-строки, для передачи ее в функцию достаточно передать указатель на ее начало. Любой указатель на символ некоторой си-строки является указателем на ее подстроку — тоже си-строку (суффикс).

С точки зрения алгебраических типов данных, си-строку можно считать рекурсивным определением (псевдокод):

```
Nul = char(0);
Cstr = (char, Cstr) | Nul;
```

Очевидно, что си-строка не может содержать символы с кодом нуль. В то же время, если по каким-то причинам завершающий нулевой символ будет потерян, то попытки работы с такой строкой, вероятно, приведут к выходу за пределы выделенного под нее блока памяти, что чревато «падением» приложения или серьезными ошибками в его работе (в том числе, риском утечки данных).

Си-строки и особенности работы с ними являются одним из основных источников «дыр» в безопасности сетевых приложений, написанных на C, поэтому в программах на C++ рекомендуется при возможности использовалась std::string или другие аналогичные возможности из сторонних библиотек.

13.1. Функционал cstring

Заголовочный файл **cstring** содержит базовый функционал для работы с си-строками и массивами символов (байт). Массивы байт передаются парой (*указатель* типа **void***, *размер* в байтах). Далее приведены краткие описания некоторых стандартных функций, объявленных в этом заголовочном файле.

Для краткости даны объявления функций только для языка C. В случае C++ нередко существуют аналогичные пары функций с различной константностью (C «теряет» константность), например, в C:

```
void * memchr(
  void const * mem, int byte, size_t count);
```

теряет свойство «только для чтения» блока mem, возвращая указатель без **const**.

В то же время в C++ (в пространстве имен std) имеется сразу два варианта:

```
void * memchr(
  void * mem, int byte, size_t count);
void const * memchr(
  void const * mem, int byte, size_t count);
```

Блоки памяти

```
void * memset(
  void * dest, int byte, size_t count);
```

заполняет блок памяти с адресом dest длиной count байт значением byte (приведенным к **unsigned char**). Возвращает dest.

```
void * memcpy(
  void * dest, void const * src, size_t count);
```

копирует содержимое блока памяти с адресом src длиной count байт в блок по адресу dest. Возвращает dest. В случае, если диапазоны [src, src+count) и [dest, dest+count) имеют непустое пересечение, поведение не определено.

```
void * memmove(
  void * dest, void const * src, size_t count);
```

действует аналогично memcpy, но также обрабатывает случай пересекающихся диапазонов. Семантика в этом случае эквивалентна копированию через временный промежуточный буфер. Наличие двух разных функций memcpy и memmove связано с тем фактом, что memcpy может быть быстрее благодаря предположению, что диапазоны не пересекаются.

```
void * memchr(
  void const * mem, int byte, size_t count);
```

ищет в блоке памяти с адресом mem длиной count байт первое с начала блока вхождение байта byte (как **unsigned char**). Если

такой байт найден, то функция возвращает его адрес. В противном случае возвращает нулевой указатель.

```
int memcmp(
  void const * lhs, void const * rhs, size_t count);
```

сравнивает побайтно лексикографически («словарно») два блока памяти (с адресами lhs и rhs) одинаковой длины count байт. Возвращает:

- значение меньше нуля, если блок lhs стоял бы в словаре до rhs;
- нуль, если блоки равны;
- значение больше нуля, если бы блок lhs стоял бы в словаре после rhs.

При сравнении порядок байт относительно друг друга определяется их числовыми значениями, а не смыслом соответствующих им символов.

Си-строки

Все описанные здесь функции порождают неопределенное поведение, если в качестве параметра, подразумевающего передачу адреса си-строки с завершающим нулем, передается указатель, не указывающий на си-строку.

```
size_t strlen(char const * str);
```

возвращает длину си-строки с адресом str — расстояние от str до первого вхождения нулевого символа, измеренное в байтах.

У Определите strlen через memchr.

```
char * strcpy(char * dest, char const * src);
```

копирует строку с адресом src (вместе с завершающим нулем) в буфер по адресу dest, возвращает dest. Поведение не определено, если выделенный буфер недостаточно большой (выделен слишком маленький блок памяти либо диапазон области копирования пересекается с исходной строкой).

```
char * strcat(char * dest, char const * src);
```

выполняет конкатенацию строк: дописывает копию си-строки по адресу src к си-строке по адресу dest. Блок памяти, в котором хранится строка по адресу dest, должен иметь достаточный размер, чтобы вместить результат конкатенации. Возвращает dest.

Наивное использование strcat может приводить к низкому быстродействию программы. Следующий код требует квадратичного времени вместо необходимого линейного:

```
// Конкатенация си-строк (квадратичное время).
char * join(char const * const * a, size_t n) {
  // Определить суммарный размер.
  size_t sz = 0;
  for (size_t i = 0; i < n; ++i)
    sz += strlen(a[i]);
  // Создать хранилище.
  char * s = (char*)malloc(sz+1);
  if (!s) return s; // не удалось создать
  s[0] = '\0';
  // Цикл конкатенации-накопления.
  for (size_t i = 0; i < n; ++i)
    strcat(s, a[i]);
  return s;
}
```

Проблема заключается в том, что strcat каждый раз проходит по накопленной строке, чтобы найти ее конец. Код можно сделать линейным, если запоминать позицию конца и выполнять простое копирование:

```
// Конкатенация си-строк (линейное время).
char * join(char const * const * a, size_t n) {
  // Определить суммарный размер.
  size_t sz = 0;
  for (size_t i = 0; i < n; ++i)
    sz += strlen(a[i]);
  // Создать хранилище.
  char * s = (char*)malloc(sz+1);
  if (!s) return s; // не удалось создать
```

```
  auto d = s; // текущая позиция записи
  // Цикл конкатенации-накопления.
  for (size_t i = 0; i < n; ++i) {
    size_t ailen = strlen(a[i]);
    memcpy(d, a[i], ailen);
    d += ailen;
  }
  s[sz] = '\0'; // завершающий нуль
  return s;
}
```

```
char * strchr(char const * str, int byte);
```

ищет в си-строке по адресу str первое вхождение байта со значением byte. Возвращает указатель на найденный байт либо нулевой указатель, если найти заданный байт не удалось.

Данную функцию часто используют для проверки принадлежности символа некоторому множеству, например, так:

```
bool is_punct(char ch) {
  return strchr(",.!?-:'\"", ch) != nullptr;
}
```

```
char * strrchr(char const * str, int byte);
```

ищет в си-строке по адресу str последнее вхождение байта со значением byte. Возвращает указатель на найденный байт либо нулевой указатель, если найти заданный байт не удалось.

```
int strcmp(char const * lhs, char const * rhs);
```

сравнивает две си-строки лексикографически.

Аналогично memcmp возвращает значение меньше нуля, если lhs стоит в словаре до rhs; значение больше нуля, если lhs стоит в словаре после rhs; нуль, если строки совпадают. При сравнении порядок байт относительно друг друга определяется их числовыми значениями (кодами символов), а не смыслом соответствующих им символов.

```
char* strstr(char const * str, char const * substr);
```

ищет в си-строке по адресу str первое вхождение подстроки, заданной си-строкой с адресом substr. Возвращает указатель на первый символ найденной подстроки либо нулевой указатель, если такой подстроки нет.

У Реализуйте аналог strstr на основе цикла, вызывающего strchr и memcmp.

13.2. Функционал cstlib

В данном заголовочном файле определены функции strtol, strtoll, strtoul, strtoull, strtof, strtod, strtold, предназначенные для распознавания записей чисел. Принцип работы данных функций аналогичен принципу работы функций stol, stoll и т. д. с той разницей, что те функции принимают string и бросают исключения (см. с. 187).

В качестве примера приведем прототипы strtol и strtof. Прочие функции отличаются от них типом возвращаемого значения:

```
long strtol(char const * in, char ** end, int base);
float strtof(char const * in, char ** end);
```

По адресу end записывается адрес символа, на котором разбор был прекращен (первый символ, не входящий в запись числа).

Определены также функции atoi, atol, atoll, по сути, являющиеся обертками stol и stoll (предполагается base = 10), предлагающие упрощенный синтаксис (передается только строка):

```
int atoi(char const * in);
```

13.3. Функционал cstdio

Данный заголовочный файл содержит различный функционал, в частности, для работы с файлами (часть стандартной библиотеки C). Однако в данном разделе мы рассмотрим

лишь определенные здесь функции форматированного ввода-вывода.

Форматированный вывод

Функции

```
int printf(char const * format,...);
int fprintf(FILE * stream, char const * format,...);
int sprintf(char * buffer, char const * format,...);
int snprintf(char * buffer, size_t buf_size,
             char const * format,...);
```

осуществляют форматированный вывод: printf — в стандартный поток вывода, fprintf — в файловый поток C, sprintf и snprintf — в массив символов (результат записи — си-строка). Вызов printf(*что-то*) эквивалентен вызову fprintf(stdin, *что-то*), stdin — указатель на объект стандартного потока C (объявлен в **cstdio**).

Функции sprintf и snprintf отличаются друг от друга тем, что первая полагает, что буфер «достаточного размера», а вторая принимает размер буфера и не допускает выход за его пределы. Конечно, в общем случае рекомендуется использовать именно второй вариант. Кроме того, snprintf можно использовать для того, чтобы определить необходимый размер буфера: она вернет его (не включая нулевой символ), если передать в качестве буфера нулевой указатель и нулевой размер.

Все варианты в случае успеха возвращают число записанных символов (не включая нулевой символ для строковых вариантов), а в случае неудачи — некоторое отрицательное значение.

Строка format определяет структуру вывода. По сути, в файл или строку выводится ее содержимое, в котором определенные подстроки заменяются на представление значений, переданных после format.

Вызов printf для format, не содержащей особых подстрок, просто выведет строку:

```
printf("Hollow world\n");
// выведет Hollow world и перевод строки.
```

Эти «особые подстроки» всегда начинаются на знак %. Для вывода самого знака % его требуется экранировать (им самим):

```
printf("%%"); // выведет %
```

Иные варианты предполагают использование очередного аргумента, переданного после format, и подстановку его представления. Полное описание возможностей printf (учитывая также имеющиеся различия между популярными реализациями) весьма громоздко, поэтому здесь представлен сокращенный вариант.

- %c: вывести символ (аргумент типа **int**);
- %s: вывести си-строку (аргумент типа **char***);
- %d или %i: вывести целое со знаком в десятичной системе. Варианты:
  - %hhd: аргумент типа **signed char**;
  - %hd: аргумент типа **short**;
  - %d: аргумент типа **int**;
  - %ld: аргумент типа **long**;
  - %lld: аргумент типа **long long**;
  - %td: аргумент типа ptrdiff_t;
  - %jd: аргумент типа intmax_t;
- %u: вывести беззнаковое целое в десятичной системе (модификаторы типа аналогичны варианту %d, но используются **unsigned**-варианты);
- %o: вывести беззнаковое целое в восьмеричной системе (модификаторы типа аналогичны варианту %u);

- %x или %X: вывести беззнаковое целое в шестнадцатеричной системе (модификаторы типа аналогичны варианту %u);

- %f или %F: вывести число с плавающей запятой в десятичной системе, %f выводит специальные значения строчными буквами (nan, inf), %F — заглавными (NAN, INF). Варианты:

  – %f и %lf: аргумент типа **double**;

  – %Lf: аргумент типа **long double**;

- %e (%E): аналогично %f (%F), но обязательно с порядком через знак e или E соответственно;

- %g (%G): вывести число с плавающей запятой аналогично %f (%F) или %e (%E), в зависимости от величины числа и выбранной точности (см. далее);

- %a (%A): аналогично %e (%E), но в шестнадцатеричной системе;

- %p: вывести указатель (аргумент типа **void***);

- %n: вернуть число записанных символов через переданный указатель на целое (модификаторы как у %d), подстрока заменяется на пустую.

```
int apples = 10, letter = 'a';
double half = 0.5;
ptrdiff_t written = 0;
printf("We have %d (%f) apples under '%c'.%tn",
       apples, half, letter, &written);
// > We have 10 (0.500000) apples under 'a'.
// written == 39
```

Другие возможные модификаторы (ставятся в порядке перечисления в списке после %, но перед буквами формата из предыдущего списка):

- −: выравнивать по левому краю (по умолчанию по правому);
- +: обязательно ставить знак (по умолчанию выводится только знак −);
- (пробел): ставить пробел, если нет знака или результат пуст, игнорируется, если есть модификатор +;
- #: использовать «альтернативное представление». Поведение не определено, если для данного формата не предусмотрено альтернативное представление. А предусмотрено оно для форматов:
  - %o: обязательно выводить ведущий 0;
  - %x (X): обязательно выводить префикс 0x (0X);
  - %f, %F, %e, %E, %g, %G %a, %A: обязательно выводить разделитель целой и дробной части (точку). В случае %g и %G выводятся замыкающие нули после точки (по умолчанию нет);
- 0: заполнять слева нулями, а не пробелами;
- целое число или ∗: минимальная ширина представления в символах. Знак ∗ означает «прочитать ширину из следующего аргумента типа **int**»;
- . (и, возможно, далее целое число или ∗): задает «точность» представления в цифрах (по умолчанию 6 для чисел с плавающей запятой). Знак ∗ означает «прочитать точность из следующего аргумента типа **int**». В случае отсутствия числа выбирается точность 0.

```
long n = -1;
double half = 0.5;
printf("(%-4ld)␣(%04ld)␣(%.3E)",
        n, n, half);
// > (-1 ) (-001) (5.000E-01)
```

Ввиду того, что функция printf, по сути, напрямую обращается к стеку вызовов для извлечения или записи значений, она является одним из основных источников «дыр безопасности» в программах, написанных на C, позволяя извлекать или изменять значения прямо в стеке вызовов (в частности, локальные переменные и адреса возврата функций). Так получается из-за того, что в качестве format передается целиком или полностью строка, полученная извне, в которую атакующему удается вставить %-директивы. Помимо этого, зависимость формата от типа и последовательности параметров увеличивает вероятность ошибок (с потенциальным неопределенным поведением в случае ошибки). В программах на C++ рекомендуется использовать стандартные потоки ввода-вывода и другие средства, которые лишены опасностей printf. Если же используется printf и строка format содержит элементы, полученные извне, то ее следует обезопасить, выполнив удвоение символов %.

Форматированный ввод

Функции

```
int scanf(char const * format,...);
int fscanf(FILE * stream, char const * format,...);
int sscanf(char const * buffer,
           char const * format,...);
```

осуществляют форматированный ввод: scanf — из стандартного потока ввода, fscanf — из файлового потока C, sscanf — из систроки.

Принцип работы scanf напоминает printf: функция разбирает format и сопоставляет подстроки с подстроками ввода, при необходимости распознавая представления чисел и записывая их в переменные, указатели на которые переданы аргументами после format.

Подстроки, требующие распознавания, начинаются на %. Подстрока %% соответствует одному знаку % во вводе. Пробельные символы соответствуют произвольной последовательности пробельных символов. Прочие символы сопоставляются

с содержимым ввода. Распознавание успешно, если они совпадают.

- %c: прочитать символ (аргумент типа **char***);
- %s: прочитать последовательность непробельных символов (аргумент типа **char***, записывает в конец завершающий нулевой символ);
- %[*символы*]: прочитать непустую последовательность, состоящую из перечисленных символов (аргумент типа **char***, записывает в конец завершающий нулевой символ). Данный вариант имеет ряд особенностей (в том числе, определяемых реализацией), поэтому в случае его использования следует воспользоваться документацией;
- %d: прочитать целое число (может быть со знаком) в десятичной форме. Варианты типов аргумента:
  - %hhd: **signed char*** или **unsigned char***;
  - %hd: **short*** или **unsigned short***;
  - %d: **int*** или **unsigned***;
  - %ld: **long*** или **unsigned long***;
  - %lld: **long long*** или **unsigned long long***;
  - %td: ptrdiff_t*;
  - %zd: size_t*;
  - %jd: intmax_t* или uintmax_t*;
- %i: прочитать целое число с возможным префиксом системы счисления (0, 0x, 0X). Варианты типов аналогично %d;
- %u: прочитать беззнаковое целое число в десятичной форме. Варианты типов аргумента:
  - %hhu: **unsigned char***;

  - %hu: **unsigned short***;
  - %u: **unsigned***;
  - %lu: **unsigned long***;
  - %llu: **unsigned long long***;
  - %tu: ptrdiff_t*;
  - %zu: size_t*;
  - %ju: uintmax_t*;

- %o: прочитать беззнаковое целое число в восьмеричной форме. Варианты типов аналогично %u;

- %x или %X прочитать беззнаковое целое число в шестнадцатеричной форме. Варианты типов аналогично %u;

- %f, %F, %e, %E, %g, %G, %a, %A: прочитать число с плавающей запятой. Варианты (на примере %f):

  - %f: аргумент типа **float***;
  - %lf: аргумент типа **double***;
  - %Lf: аргумент типа **long double***;

- %p: прочитать указатель (как если бы он был выведен функцией printf), аргумент типа **void****;

- %n: вернуть число прочитанных символов. Варианты типов аналогично %d.

Аналогично printf, между % и символами типа можно указать дополнительные опции формата (в порядке перечисления):

- *: пропустить значение, не записывая его в переменную (аргумент не поглощается);

- целое число, большее нуля: максимальная ширина поля. Данный параметр желательно задавать при использовании %s, иначе есть риск переполнения буфера. При использовании с %c записывает заданное число символов в буфер по переданному адресу (без завершающего нуля). Например: %3c — прочитать ровно три символа.

Функции scanf возвращают число успешно распознанных и записанных значений или EOF (обычно −1, эта константа определена в **cstdio**), если не удалось приступить к разбору представления первого (имеющегося) аргумента или если не удалось прочитать ни одного символа (например, файл кончился). В противном случае, если дополнительных аргументов нет, то результат будет 0, даже если не удалось сопоставить ни одного символа из строки format.

Чтобы прочитать первый непробельный символ, следует поставить пробел перед %c:

```
// Любой символ.
char get_ch() {
  char val = 0;
  scanf("%c", &val);
  return val;
}
// Первый непробельный символ.
char ws_ch() {
  char val = 0;
  scanf(" %c", &val);
  return val;
}
```

13.4. Функционал cctype

Данный заголовочный файл предоставляет набор простых функций, позволяющий классифицировать и преобразовывать символы базовой кодировки.

Все функции, перечисленные в этом разделе, принимают в качестве кода символа значение типа **int**. В качестве этого значения может быть передан код символа (приведенный к **unsigned char**, поскольку если **char** знаковый, то значения могут попасть в отрицательную область и привести к неопределенному поведению) или специальное значение «символ конца файла» EOF.

Классификация символов

- isalpha — является ли буквой;
- isdigit — является ли десятичной цифрой;
- isalnum — является ли буквой или цифрой;
- islower — является ли строчной буквой;
- isupper — является ли заглавной буквой;
- isxdigit — является ли шестнадцатеричной цифрой;
- isspace — является ли пробельным символом;
- isblank — более узкая версия isspace: проверяет, является ли символ пробелом или горизонтальной табуляцией;
- isprint — является ли печатаемым символом;
- iscntrl — является ли управляющим символом (отрицание isprint);
- isgraph — является ли графическим символом (включает все печатные символы, имеющие изображение, т. е. исключает пробел).

Преобразование символов

Определены две функции: tolower и toupper. Первая приводит заглавные буквы к их строчным аналогам, вторая — наоборот, строчные к заглавным. Если передать код другого символа, то он будет возвращен без изменений. Тип возвращаемого значения — **int**.

13.5. Тип string_view

Представим ситуацию: надо определить функцию, принимающую строку для чтения. Можно использовать std::string:

```
bool foo(string const & s);
```

Однако в случае передачи такой функции си-строки (в том числе, строковой константы) будет создан новый временный объект string с копией переданной строки:

```
if (foo("an example"))
  // ...
```

Это, конечно, работает, но возникает желание избежать избыточного копирования данных.

Вместо string мы можем принимать си-строку:

```
bool foo(char const * s);
```

Чтобы передать такой функции содержимое объекта string, следует воспользоваться функцией c_str:

```
string answer;
if (getline(cin, answer) && foo(answer.c_str()))
  // ...
```

В качестве преимущества мы получаем совместимость с другим кодом на C или других языках программирования, предоставляющих средства для работы с си-строками, или даже с кодом на C++, который не использует стандартные строки. Но писать качественный код, работающий с си-строками, сложнее, чем аналогичный код, использующий string.

В качестве возможного решения подобных проблем, а также средства, упрощающего работу со строками (в том числе, си-строками), стандартная библиотека C++17 предоставляет еще один класс: std::string_view, определенный в заголовочном файле **string_view**.

Функции, принимающей string_view, можно передавать как объекты string, так и си-строки, никакого копирования данных при этом не происходит:

```
bool foo(string_view s);
```

Дело в том, что string_view представляет собой попросту пару *указатель на массив символов*, *размер массива символов*. При инициализации объектом string эту пару можно извлечь вызовами функций data и size, соответственно. При инициализации си-строкой передается как раз указатель, а размер можно получить вызовом strlen для этого указателя (один раз пройтись по строке в поиске завершающего нуля).

В целом, использовать объект string_view можно как ссылку на string **const**: предоставляется аналогичный набор операций. Любые корректные[117] действия с string_view не изменяют строку, из которой он создан.

Подробное рассмотрение string_view выходит за рамки данной книги, но отметим три предоставляемых функции:

- remove_prefix(n) — отрезать n символов с начала строки (передвижением указателя вперед на n позиций). Если $n >$ размер, то поведение не определено.

- remove_suffix(n) — отрезать n символов с конца строки (уменьшением размера на n). Если $n >$ размер, то поведение не определено.

- substr($p = 0$, $c =$ npos) возвращает объект string_view, соответствующий подстроке, начинающейся в позиции p и

[117] Мы всегда можем получить указатель на символ, снять модификатор **const** и попытаться изменить строку (UB).

имеющей длину c. Если c не передать или передать c большее, чем длина суффикса строки, то будет возвращен суффикс строки с позиции p. Если $p \geqslant$ размер, то функция бросает исключение.

Потенциальная проблема при использовании string_view, в общем-то, та же, что при использовании ссылок на string и указателей на си-строки: исходные строки должны существовать в течение всего времени использования. Поэтому не рекомендуется хранить объекты string_view, так как вполне можно попасть в неприятную ситуацию, когда строка, на которую указывает хранимый string_view, уже не существует.

Задания для самопроверки

У Ответьте на следующие вопросы.

- Какие символы не может хранить си-строка?
- В чем отличие функции memmove от функции memcpy?
- В чем отличие функции strcpy от функции memcpy?
- Как с помощью printf вывести знак процента, не использовав в коде символ % ни разу?
- Чем при использовании printf отличается формат %e от формата %A?
- Как с помощью printf узнать длину выводимого текста?
- Что возвращает функция scanf?
- Какой функцией можно проверить, что символ является буквой или цифрой?
- Почему может быть нужно приводить аргумент функций cctype к **unsigned char**? Например, так:

```
if (char ch; cin.get(ch)
    && isalpha((unsigned char)ch))
  // ...
```

- В чем отличие функции isspace от функции isblank?
- В чем отличие функции isprint от функции isgraph?

У Запишите с помощью scanf чтение вектора в формате (x, y), где x и y имеют тип **double**.

У Определите следующие функции:

```
// Убрать максимальный префикс из пробельных символов.
string_view ltrim(string_view);

// Убрать максимальный суффикс из пробельных символов.
string_view rtrim(string_view);

// Убрать максимальные и префикс, и суффикс
// из пробельных символов.
string_view trim(string_view);
```

У Определите следующие функции:

```
// Определить длину максимального общего префикса.
size_t max_common_prefix
    (string_view a, string_view b);

// Определить, является ли pre префиксом s.
bool begins_with(string_view s, string_view pre);
```

Глава 14

Инварианты и разработка программ

14.1. Инварианты

Предусловие [*precondition*] — предикат, истинный на всех корректных кортежах значений параметров функции. Нарушение предусловия означает неверное использование функции.
Постусловие [*postcondition*] — предикат, истинный для всех корректных результатов функции. Нередко постусловия зависят также от параметров функции. Нарушение постусловия означает ошибку внутри функции.
Инвариант цикла [*loop invariant*] — предикат, истинный на некотором наборе переменных перед входом в цикл и после каждой итерации цикла. Инвариант цикла может нарушаться внутри итерации цикла.
Инвариант класса [*class invariant*] — предикат, истинный для объектов этого класса между вызовами функций-членов. Инвариант класса может нарушаться внутри функции-члена. Таким образом, инварианты класса являются пред- и постусловиями функций-членов.

Выявление инвариантов циклов и классов используется при доказательстве корректности программы.

Стандартные средства поддержки проверки предикатов в C++ включают два элемента: ключевое слово **static_assert** и макрос[118] assert, определенный в заголовочном файле **cassert**.

Первое задает предикат, проверяемый во время компиляции. Если этот предикат не является константой времени компиляции или ложен, компилятор сообщает об ошибке компиляции:

```
// Не компилируется, если размер указателя не равен 8 байтам:
static_assert(sizeof(void*) == 8);
```

В качестве необязательного второго параметра можно указать сообщение об ошибке, которое будет выведено компилятором, если предикат окажется ложным:

```
static_assert(sizeof(void*) == 8,
      "Program needs a 64-bit CPU.");
// В случае, если размер указателя не равен 8 байтам,
// компилятор выведет Program needs a 64-bit CPU.
```

Проверка **static_assert** может быть размещена в тех местах программы, где может быть размещено определение. Например, в определении класса или теле функции, но не в выражении и не в списке параметров функции.

Макрос assert предназначен для проверки условий во время исполнения и принимает один аргумент. Если значение этого аргумента оказывается ложным, в консоль (а точнее, в поток ошибок) выводится сообщение об ошибке с указанием места сработавшего условия, и работа программы прекращается.

```
float lsolve(float a, float b) {
  assert(a != 0);
  return -b / a;
}
```

Конструкция assert(*аргумент*) является выражением и может размещаться в тех местах, где допустимо выражение. Как правило, assert используется в виде инструкции (как в примерах в этой главе).

[118] Подробнее о макросах см. в разд. 18.2.

Необходимо отметить, что assert может вовсе ничего не делать (даже выражение, размещенное в качестве его аргумента, может не вычисляться), поскольку может быть «выключен» настройками компилятора. Как правило, assert выключен, если определен макрос NDEBUG («нет отладки»), и включен в противном случае.

Иногда assert используется для того, чтобы «закрыть» секцию **default** в конструкции **switch-case**, где учтены все возможные случаи:

```
enum Color { RED, YELLOW, GREEN };
char const * day_of_week(Color color) {
  switch (color) {
  case RED:    return "red";
  case YELLOW: return "yellow";
  case GREEN:  return "green";
  default:
    assert(!"Impossible color value!");
    return "unknown";
}
```

В данном примере можно отметить два момента.

Во-первых, здесь аргумент assert всегда ложен, поскольку это отрицание указателя на размещенный в памяти констант массив символов. Ошибкой является сам факт достижения исполнителем данной строки программы, а поскольку условие сработавшего assert выводится в консоль, мы увидим эту строку в качестве сообщения об ошибке.

Во-вторых, функция все равно должна сделать что-то максимально осмысленное после assert, поскольку он может быть выключен, и в этом случае исполнение программы продолжится. В примере программа в любом случае возвращает указатель на корректную строку.

О реализации макросов, аналогичных assert, см. на с. 438.

В качестве примера рассмотрим следующую задачу: вычислить массу молекулы химического вещества по его формуле.

Масса вычисляется в атомных единицах массы, которые берутся из таблицы. Положим, что таблица хранится в тексто-

вом файле elements.table и представляет собой последовательность пар «название элемента — масса» (все разделители — пробельные символы). Например, таблица биологических макроэлементов может иметь следующий вид:

```
H       1.008
C       12.01
N       14.007
O       16
Na      22.99
Mg      24.306
P       30.974
S       32.068
Cl      35.45
K       39.098
Ca      40.078
```

Название элемента либо состоит из одной заглавной латинской буквы, либо состоит из двух латинских букв, первая из которых заглавная, а вторая — строчная.

Формула представляет собой последовательность названий элементов или взятых в круглые скобки групп, после которых может идти натуральное число — количество атомов элемента или повторений группы. Собственно группа — это тоже формула. Пробелы между частями формулы игнорируются. Внутри названий элементов или записей чисел пробелов быть не может.

Далее приведено несколько формул с указанием масс (соответственно приведенной ранее таблице), которые можно использовать для проверки программы (пробелы оставить).

Формула	Масса	Формула	Масса
`H2`	2.016	`C60`	720.6
`C6H6`	78.108	`KCN`	65.115
`H 2 O`	18.016	`H2 SO4`	98.084
`CaCl2`	110.978	`Na 3 PO 4`	163.944
`(OH)2`	34.016	`Mg(OH)2`	58.322
`CH3 CH2 OH`	46.068	`( CH2 )2`	28.052

Вначале перечислим используемые заголовочные файлы:

Пример 14.1. CheMass часть 1

```
#include <cctype>
#include <string>
#include <string_view>
#include <fstream>
#include <iostream>
using namespace std;
```

Часть два содержит описание глобальных переменных. В данном случае это параллельные массивы масс mass и имен name элементов, а также количество прочитанных элементов elems, которое не может быть больше размера массивов — константы MAX_ELEMS.

Пример 14.2. CheMass часть 2

```
const int MAX_ELEMS = 200;
double mass[MAX_ELEMS];
string name[MAX_ELEMS];
int elems;
```

Часть три содержит точку входа и описывает логику работы консольного приложения: сначала прочитаем описание таблицы (заполним глобальное состояние), затем будем построчно считывать формулы и выводить вычисленные массы.

Пример 14.3. CheMass часть 3

```
void read_elems();
double compound_mass(string_view formula);

int main() {
  read_elems();
  for (string line; getline(cin, line);)
    cout << compound_mass(line) << '\n';
  return 0;
}
```

Часть четыре — чтение таблицы элементов read_elems и извлечение массы элемента по его названию name_to_mass.

У Что случится, если файл не удастся открыть или его содержимое будет иметь неверный формат?

Пример 14.4. CheMass часть 4

```
// Считать таблицу элементов.
void read_elems() {
  ifstream table("elements.table");
  string elem_name;
  double elem_mass;
  while (elems < MAX_ELEMS &&
    table >> elem_name >> elem_mass) {
    name[elems] = elem_name;
    mass[elems] = elem_mass;
    ++elems;
  }
}

// Определить массу по названию.
// Возвращает 0, если название не найдено.
double name_to_mass(string_view elem) {
  for (int i = 0; i < elems; ++i)
    if (name[i] == elem)
      return mass[i];
  return 0;
}
```

Часть пять содержит алгоритм разбора формулы и вычисления суммарной массы. В случае обнаружения скобочной формы (открывающей скобки) производится поиск закрывающей скобки, и содержимое между скобками интерпретируется рекурсивно как отдельная формула. Данная функция использует несколько вспомогательных функций, определенных далее.

Пример 14.5. CheMass часть 5

```
int blank_prefix(string_view formula);
string_view get_int(string_view formula);
int str_to_int(string_view int_view);
string_view get_name(string_view formula);
```

```
int find_close(string_view formula);

// Вычислить массу вещества по формуле.
double compound_mass(string_view formula) {
  double last_mass = 0, accum_mass = 0;
  string_view name_pre, int_pre;
  for (;;) {
    // Убрать все пробелы.
    formula.remove_prefix(blank_prefix(formula));
    if (formula.empty())
      break;

    // Открывающая скобка?
    if (formula[0] == '(') {
      int pre = find_close(formula);
      last_mass = compound_mass
        (formula.substr(1, pre - 1));
      formula.remove_prefix(pre);
    }
    else {
      // Определить элемент.
      name_pre = get_name(formula);
      last_mass = name_to_mass(name_pre);
      formula.remove_prefix(name_pre.size());
    }

    // Определить количество.
    int_pre = get_int(formula);
    int how_many = str_to_int(int_pre);
    last_mass *= how_many;
    formula.remove_prefix(int_pre.size());

    // Элемент не найден — выход.
    if (last_mass == 0)
      break;

    accum_mass += last_mass;
  }
  return accum_mass;
}
```

Часть шесть — функция, определяющая, сколько пробельных символов стоит в начале переданной ей строки.

Пример 14.6. CheMass часть 6

```
// Вычислить длину префикса, состоящего из пробелов.
int blank_prefix(string_view formula) {
  int i = 0, sz = formula.size();
  while (i < sz && isspace(formula[i]))
    ++i;
  return i;
}
```

Часть семь — две функции, используемые для извлечения чисел. Первая функция определяет, сколько цифр стоит в начале переданной ей строки. Вторая функция интерпретирует запись числа.

Пример 14.7. CheMass часть 7

```
// Извлечь из строки префикс-целое число.
// Возвращает пустой, если нет префикса-числа.
string_view get_int(string_view formula) {
  int i = 0, sz = formula.size();
  while (i < sz && isdigit(formula[i]))
    ++i;
  return formula.substr(0, i);
}

// Строковое представление числа → число.
int str_to_int(string_view int_view) {
  int result = 0;
  for (int i = int_view.size(); i != 0; --i)
    result = 10*result + (int_view[i-1] - '0');
  return result;
}
```

Часть восемь — функция, пытающаяся определить название элемента, записанное в начале строки. Оно может состоять из одной или двух букв.

Пример 14.8. CheMass часть 8

```
// Извлечь из строки префикс — название элемента.
// Возвращает пустой, если нет префикса-названия.
string_view get_name(string_view formula) {
  int len = 0;
  if (isupper(formula[0])) {
    len = 1;
    if (islower(formula[1]))
      len = 2;
  }
  return formula.substr(0, len);
}
```

И наконец, последняя, девятая часть — функция, ищущая конец скобочной формы. Для этого используется простой алгоритм: очередная открывающая скобка увеличивает счетчик уровня вложенности на единицу, а закрывающая скобка уменьшает счетчик на единицу. Если счетчик достиг нуля, то, значит, скобочная форма закончилась.

Пример 14.9. CheMass часть 9

```
// Найти позицию за закрывающей скобкой.
// Предполагается, что formula[0] == '('
int find_close(string_view formula) {
  int i = 1, sz = formula.size(), d = 1;
  for (; i < sz && d != 0; ++i)
    switch (formula[i]) {
    case '(': ++d;
    case ')': --d;
    default:; // игнорируем другие символы
    }
  return i;
}
```

Если запустить приведенный выше код, то легко убедиться, что он содержит ошибки. Кроме того, он не уведомляет пользователя об ошибках ввода-вывода.

Попробуем для каждой функции определить предусловия и постусловия.

Функция main. Данная функция не принимает параметров и всегда возвращает 0, соответственно, никаких предусловий и постусловий здесь нет. В функции есть цикл **for**, считывающий ввод из стандартного потока ввода по строкам, инвариантом является «line содержит очередную успешно считанную строку».

Функция read_elems. Данная функция не принимает параметров, а ее результатом фактически являются заполненные таблицы name и mass и значение elems. Соответственно, можно выделить постусловия:

- elems $\leqslant$ MAX_ELEMS — истинно, если истинно в качестве предусловия (тогда оно становится инвариантом цикла **while** в этой функции).
- elems = размеру заполненной области name и mass — истинно как инвариант цикла **while**.
- Для любого $0 \leqslant i <$ elems name[i] — корректное название элемента и mass[i] — корректная атомная масса элемента. Проверку данного постусловия можно вставить непосредственно в цикл. То, что используемые название и массу удалось считать из файла, гарантируется условием цикла (иначе тело не выполнится).

Также можно выделить предусловие «файл elements.table существует».

В результате получим следующий измененный код:

Пример 14.10. CheMass-b read_elems

```
bool is_correct_name(string_view name) {
  return (name.size() == 1 && isupper(name[0])) ||
         (name.size() == 2 && isupper(name[0])
                           && islower(name[1]));
}

// Считать таблицу элементов.
void read_elems() {
```

```
    assert(elems <= MAX_ELEMS);
    ifstream table("elements.table");
    assert(table.is_open());
    string elem_name;
    double elem_mass;
    while (elems < MAX_ELEMS &&
           table >> elem_name >> elem_mass) {
      assert(is_correct_name(elem_name));
      assert(elem_mass > 0);
      name[elems] = elem_name;
      mass[elems] = elem_mass;
      ++elems;
    }
}
```

Здесь мы воспользовались макросом assert, определенным в стандартной библиотеке. Это простейшее доступное средство для проверки условий корректности в коде. Для того, чтобы его использовать, требуется подключить соответствующий заголовочный файл:

```
#include <cassert>
```

Функция name_to_mass. Данная функция попросту перебирает все названия в попытке найти совпадающее с заданным. Можно выделить предусловия elems $\leqslant$ MAX_ELEMS и is_correct_name(elem).

Функция compound_mass. Эта функция принимает формулу и возвращает массу. Исходя из этого, можно посчитать, что предусловие есть «формула корректна», а постусловие — «масса корректна». Однако это не так. Во-первых, проверка корректности формулы, по сути, эквивалентна ее разбору, что как раз выполняется данной функцией. Во-вторых, строка с формулой приходит «извне» (от пользователя) и может содержать что угодно. Поэтому никаких предусловий накладывать не будем. Корректность формулы желательно проверять в процессе разбора, сообщая об обнаруженных ошибках.

Что до корректности результата в предположении корректности формулы, то это не постусловие, это следствие коррект-

ности кода. Можно сказать, что это теорема, которую надо доказывать. Итак, постусловий тоже нет.

Функция blank_prefix. Данная функция нужна для выбрасывания пробельных символов. Предусловий нет. Постусловие: либо i = длине formula, либо formula[i] не является пробельным символом. Однако это условие есть не что иное, как отрицание условия продолжения цикла **while**, а значит, оно истинно после выхода из цикла и проверять его не требуется.

Функция get_int. Аналогична функции blank_prefix. Если префикс формулы не содержит цифр, то функция возвращает пустую подстроку.

Функция str_to_int. Предусловие: строка int_view не пуста и содержит только цифры. Кроме того, заметим, что из-за ограниченности диапазона **int** возможно переполнение в цикле (значение result не может убывать).

Пример 14.11. CheMass-b str_to_int

```
bool is_int_range(string_view s) {
  for (char c: s)
    if (!isdigit(c))
      return false;
  return true;
}

// Строковое представление числа → число.
int str_to_int(string_view int_view) {
assert(!int_view.empty() && is_int_range(int_view));
  int result = 0;
  for (int i = int_view.size(); i != 0; --i) {
    const int t = 10*result + (int_view[i-1] - '0');
    assert(result <= t);
    result = t;
  }
  return result;
}
```

Функция get_name. Извлекает префикс — название элемента. Постусловие: извлеченный префикс либо пуст, либо является корректным названием.

Функция find_close. Предусловие: formula не пуста и начинается на открывающую скобку. Постусловие: либо i == sz, либо formula[i−1] == ')'.

Если полученный код запустить и начать вводить формулы, то можно наткнуться на сохранившиеся ошибки первого варианта CheMass, о которых будет сигнализировать нарушение проверяемых условий.

Например, ввод формул `H 2 O`, `CaCl2` и `(OH)2` дает нарушение предусловия str_to_int. Данная функция вызывается только из одного места (строчка 96). Очевидно, проблема в том, что мы пытаемся распознать пустую строку как число. Пустая строка означает, что подразумевается число 1 (один атом). Поправим этот участок:

```
if (!int_pre.empty()) {
  int how_many = str_to_int(int_pre);
  last_mass *= how_many;
}
```

Ряд формул после этого начинает вычисляться корректно, но не `H 2 O`, которая вызывает срабатывание предусловия name_to_mass (неверное имя элемента). Данная функция также вызывается только из одного места. Судя по постусловию get_name, мы опять имеем дело с пустой строкой. Смотрим текст функции compound_mass. Поскольку вводилась формула `H 2 O`, то очевидно, что мы застряли на цифре, так как пробелы были предварительно пропущены вызовом

```
formula.remove_prefix(blank_prefix(formula));
```

Проблема заключается в том, что мы забыли пропустить пробелы перед попыткой прочитать число. Итак, настоящая ошибка находится не здесь, а дальше в коде. Добавим удаление пробелов перед чтением числа:

```
// Определить количество.
```

```
formula.remove_prefix(blank_prefix(formula)); // <
int_pre = get_int(formula);
```

С формулой `H 2 O` теперь все в порядке. Но то же самое предусловие срабатывает на формуле `(OH)2`. Посмотрим на код, отвечающий за обработку скобочной формы:

```
// Открывающая скобка?
if (formula[0] == '(') {
  int pre = find_close(formula);
  last_mass = compound_mass
    (formula.substr(1, pre - 1));
  formula.remove_prefix(pre);
}
```

Здесь pre — позиция сразу за закрывающей скобкой. Удаляя из formula префикс длины pre, мы удаляем символы с индексами 0...(pre − 1), включая саму закрывающую скобку. Но вот выделенная подстрока имеет неверный размер. Так как вся скобочная форма состоит из pre символов и мы хотим отбросить открывающую и закрывающую скобки, то длина подстроки будет (pre − 2). Исправим:

```
  last_mass = compound_mass
    (formula.substr(1, pre - 2));
}
```

Итак, проверка предусловий и постусловий непосредственно в программе позволяет значительно ускорить процесс отладки благодаря более легкой локализации ошибок и более раннему их обнаружению и чувствовать бóльшую уверенность в качестве своего кода.

Есть, впрочем, в этом коде одна ошибка, которую не поймать с помощью введенных нами предусловий и постусловий. Для того, чтобы заметить ее, достаточно попробовать ввести формулу `H10`. Окажется, что 10 интерпретируется как 1. Что-то не так с функциями get_int и str_to_int. Если ввести `H11`, то увидим правильный ответ. Если ввести `H12`, то увидим результат, соответствующий формуле H_{21}, откуда можно сделать

вывод, что виновата функция str_to_int, которая интерпретирует запись числа в обратном порядке. Итак, ошибка найдена, и исправить ее легко.

14.2. Обработка ошибок

Обработка ошибок — один из важнейших аспектов разработки программного обеспечения. Ошибки можно разбить на несколько различных видов, требующих разного подхода:

- ошибки в коде — в идеальной программе их быть не должно. Проверка предусловий и постусловий из предыдущего раздела как раз является частью стратегии выявления таких ошибок;

- ошибки программного и аппаратного обеспечения. Желательно помнить о возможности случайных ошибок в операционных системах, драйверах, библиотеках, компиляторах, о сбоях оборудования. Бороться с такими ошибками затруднительно, и нередко разработчики «умывают руки» и не предпринимают никаких мер по этому поводу;

- ошибки целостности. Имеются в виду ситуации вроде удаления или повреждения файлов, используемых программой, а также преднамеренного взлома;

- ошибки ввода. Программа, получающая извне какие-либо данные, особенно, если они вводятся человеком непосредственно, должна предусматривать выявление и обработку ошибок ввода.

Данный раздел посвящен последнему пункту. В качестве примера продолжим использовать проект CheMass.

Так как считается, что формулы вводит пользователь, требуется выявлять ошибки ввода и формировать уведомления о них. Уведомления будем выводить в поток вывода ошибок.

Сообщение об ошибке будет также включать номер введенной строки.

Первая проблема: что, если файл elements.table прочитать не удалось? Это ошибка целостности. Без информации об элементах продолжать бессмысленно, поэтому будем завершать в таком случае работу с ошибкой.

Функция read_elems. Можно предположить возможность двух ошибок: 1) файл elements.table вообще не удалось открыть; 2) из файла не удалось прочитать ни одного элемента. Соответственно, можно завести две константы — «коды ошибок»:

```
int const ERR_ELEMENTS_NOT_FOUND = 1,
          ERR_ELEMENTS_BAD_FILE  = 2;
```

На практике, однако, коды ошибок лучше оформлять в виде особого типа (который ограничивает возможные варианты этих кодов, ведь это не любое целое число!). Например, так:

Пример 14.12. Коды ошибок read_elems

```
enum Elements_table_error {
    ERR_ELEMENTS_OK,
    ERR_ELEMENTS_NOT_FOUND,
    ERR_ELEMENTS_BAD_FILE };
```

Соответственно, изменим код функции read_elems:

Пример 14.13. Функция read_elems

```
Elements_table_error read_elems() {
  assert(elems <= MAX_ELEMS);
  ifstream table("elements.table");
  if (!table.is_open())
    return ERR_ELEMENTS_NOT_FOUND;
  string elem_name;
  double elem_mass;
  while (elems < MAX_ELEMS &&
    table >> elem_name >> elem_mass) {
    if (is_correct_name(elem_name) &&
        elem_mass > 0) {
```

```
      name[elems] = elem_name;
      mass[elems] = elem_mass;
      ++elems;
    }
    else {
      clog << "Error: invalid element: "
           << elem_name << ' ' << elem_mass << '\n';
    }
  }
  if (elems == 0)
    return ERR_ELEMENTS_BAD_FILE;
  return ERR_ELEMENTS_OK;
}
```

Обратите внимание, что assert из цикла пропали, вместо этого мы либо добавляем элемент, либо выводим диагностическое сообщение.

Теперь следует добавить проверку результата read_elems в функцию main. Это легко сделать с помощью инструкции **switch**:

```
int main() {
  switch (read_elems()) {
  case ERR_ELEMENTS_OK:
    for (string line; getline(cin, line);)
      cout << compound_mass(line) << '\n';
    return 0;
  case ERR_ELEMENTS_NOT_FOUND:
    cerr << "Error: elements.table not found\n";
    return 1;
  case ERR_ELEMENTS_BAD_FILE:
    cerr << "Error: no elements have been read\n";
    return 2;
  default:
    assert(!"Impossible read_elems() result.");
    return -1;
  }
}
```

Теперь обратим внимание на строчку

```
cout << compound_mass(line) << '\n';
```

Здесь мы сразу выводим результат вычисления массы по формуле, и нет никакой возможности не делать этого в случае ошибки ввода или вообще как-то узнать о такой ошибке.

Одним из возможных решений может быть возвращать не только массу, но и код ошибки. Сделать это можно по-разному. Типичным решением является возвращение из функции кода ошибки в качестве результата, в то время как действительный результат (здесь масса) передается путем записи в переменную по ссылке.

Можно сразу предположить список возможных ошибок при разборе формулы:

- передана строка, не содержащая ничего, кроме пробельных символов;
- есть закрывающая скобка, но нет открывающей;
- есть открывающая скобка, но нет закрывающей;
- выход за пределы диапазона целого числа;
- неизвестный символ (не скобка, не цифра и не название элемента);
- неизвестный элемент.

```
enum Formula_error {
    ERR_FORMULA_OK,
    ERR_FORMULA_EMPTY,
    ERR_FORMULA_UNMATCHED_OPEN,
    ERR_FORMULA_UNMATCHED_CLOSE,
    ERR_FORMULA_INT_OVERFLOW,
    ERR_FORMULA_UNEXPECTED,
    ERR_FORMULA_UNKNOWN_ELEMENT };

// Прототип функции compound_mass.
Formula_error compound_mass(
    string_view formula, double & mass);
```

Теперь надо изменить функцию main для отражения изменения синтаксиса вызова функции compound_mass и вывода сообщений об ошибках. Каждый вывод теперь содержит номер введенной строки (формулы) num:

```
int main() {
  int num = 0;
  switch (read_elems()) {
  case ERR_ELEMENTS_OK:
    for (string line; getline(cin, line); ++num) {
      double mass = 0;
      switch (compound_mass(line, mass)) {
      case ERR_FORMULA_OK:
        cout << num << ": " << mass << '\n';
        break;
      case ERR_FORMULA_EMPTY:
        continue;
      case ERR_FORMULA_UNMATCHED_OPEN:
        clog << num << ": unmatched (\n";
        break;
      case ERR_FORMULA_UNMATCHED_CLOSE:
        clog << num << ": unmatched )\n";
        break;
      case ERR_FORMULA_INT_OVERFLOW:
        clog << num << ": integer too big\n";
        break;
      case ERR_FORMULA_UNEXPECTED:
        clog << num << ": unexpected character\n";
        break;
      case ERR_FORMULA_UNKNOWN_ELEMENT:
        clog << num << ": unknown element\n";
        break;
      default:
        assert(!"Impossible Formula_error.");
      }
    }
    return 0;
  case ERR_ELEMENTS_NOT_FOUND:
    cerr << "Error: elements.table not found\n";
    return 1;
```

```
  case ERR_ELEMENTS_BAD_FILE:
    cerr << "Error: no elements have been read\n";
    return 2;
  default:
    assert(!"Impossible read_elems() result.");
    return -1;
  }
}
```

Здесь можно остановиться и посмотреть на то, что получилось. А получилось, что код не отличается красотой и ясностью. Нагромождение **switch-case**, причем отражающих весьма регулярную схему действия из трех вариантов: нет ошибки, известная ошибка, неизвестная ошибка. Наверняка ее можно сформулировать как-то иначе. Например, вынести все сообщения об ошибках в массивы.

В случае read_elems:

```
enum Elements_table_error {
    ERR_ELEMENTS_OK,
    ERR_ELEMENTS_NOT_FOUND,
    ERR_ELEMENTS_BAD_FILE    };

string_view const elements_table_error_msg[] {
    "ok\n", // не используется
    "Fatal error: elements.table not found\n",
    "Fatal error: no elements have been read\n" };
```

Функция main принимает следующий вид (избавились от внешнего **switch**):

```
int main() {
  if (Elements_table_error e = read_elems()) {
    assert(e < elements_table_error_msg.size());
    cerr << elements_table_error_msg[e];
    return e;
  }

  int num = 0;
  for (string line; getline(cin, line); ++num)
    // тело цикла пропущено для краткости
```

```
  return 0;
}
```

Вернемся к функции compound_mass. Заметим, что старая версия функции просто возвращала 0 для «пустой строки». Это не приводило к каким-то ошибкам исполнения. Кроме того, с помощью манипулятора std::ws мы можем исключить передачу пустых строк прямо в цикле чтения из cin:

```
for (string line; getline(cin >> ws, line);
```

Поэтому ошибку «пустая строка» уберем.

Итак, имеем следующие ошибки и их описания:

```
enum Formula_error {
    ERR_FORMULA_OK,
    ERR_FORMULA_UNMATCHED_OPEN,
    ERR_FORMULA_UNMATCHED_CLOSE,
    ERR_FORMULA_INT_OVERFLOW,
    ERR_FORMULA_UNEXPECTED,
    ERR_FORMULA_UNKNOWN_ELEMENT };

string_view const formula_error_msg[] {
    "ok",
    "unmatched ( ",
    "unmatched )",
    "integer too big",
    "unexpected character",
    "unknown element" };
```

В отличие от случая с функцией read_elems, здесь хотелось бы видеть не только информацию о виде ошибки, но и хотя бы позицию символа в формуле, где эта ошибка была обнаружена. Итак, пусть функция compound_mass возвращает еще одно значение — длину успешно разобранного префикса формулы.

Новый вариант main выглядит так:

Пример 14.14. Функция main

```
Formula_error compound_mass(
    string_view formula,
    double & mass,
```

```
    string_view::size_type & chars_parsed);

int main() {
  if (Elements_table_error e = read_elems()) {
    assert(e < elements_table_error_msg.size());
    cerr << elements_table_error_msg[e];
    return e;
  }

  int num = 0;
  for (string line; getline(cin >> ws, line);
       ++num) {
    double mass = 0;
    string_view::size_type pos = 0;
    if (Formula_error e =
        compound_mass(line, mass, pos)) {
      assert(e < formula_error_msg.size());
      clog << num << ", " << pos << ": "
           << formula_error_msg[e] << endl;
      continue;
    }
    cout << num << ": " << mass << '\n';
  }
  return 0;
}
```

14.3. Метрики исходного кода

Управляющий персонал всегда хочет иметь некие простые способы оценки качества и объема работы работников. Однако чем сложнее эта работа, тем сложнее оценить эффективность ее исполнения. Если продукт — некие предметы, изготавляемые массово, то их качество можно определить различными методами, а объем работы — количеством выпущенной продукции за единицу времени. Если же продукт — программное обеспечение (ПО) или иная «интеллектуальная собственность», то все становится много сложнее. Данный продукт по природе уникален, часто требует вложения труда многих людей, и

оценить усилия и эффективность каждого участника, разделить вклады разных людей становится весьма затруднительно, особенно «со стороны». И еще сложнее оценить трудоемкость предприятия, за которое только предстоит взяться![119]

Желание иметь способы оценки трудозатрат на разработку программного обеспечения привело к появлению множества «метрик» исходного кода — алгоритмически вычислимых числовых показателей сложности. Здесь мы не будем подробно их рассматривать. О сложности оценки трудозатрат на разработку ПО и причинах этой сложности написано множество статей (и даже целые книги). Авторы приведут следующую цитату: «когда метрика становится целью, она перестает быть хорошей метрикой»[120].

В качестве простейшей метрики ПО используется суммарное число строк кода, написанных или которые требуется написать для реализации данного ПО (*lines of code*, LoC). Пустые строки, комментарии и строки, сгенерированные автоматически вспомогательными средствами, не учитываются. Условность этой метрики очевидна даже при ее «честном» применении (например, разработчиком для оценки себя самого). Тем не менее, она полезна. При разумном применении.

Примерная оценка масштаба программы при наличии единственного разработчика с субъективной точки зрения одного из авторов:

- менее 300 строк — крошечная (часы);
- менее 2'000 строк — маленькая (дни–недели);
- менее 20'000 строк — средняя (недели–месяцы);
- более 20'000 строк — большая (годы);
- более 100'000 строк — слишком большая!

[119] Одно из эмпирических правил: оцените вероятные затраты времени, затем умножьте оценку на π.

[120] См.: *Strathern M.* 'Improving ratings': audit in the British University system // European Review. 1997. № 5(3). P. 305–321.

14.4. Процесс разработки

Процесс разработки — комплекс мер и способ организации деятельности, предлагаемый или предпринимаемый в качестве средства повышения результативности разработки ПО. Различным процессам разработки (коих к данному моменту предложено великое множество) посвящен свой пласт литературы и учебных курсов, и мы воздержимся от подробного рассмотрения данного вопроса.

Стандартизация процесса разработки может потребоваться по двум причинам: для организации взаимодействия внутри команды разработчиков и достижения «чувства контроля» начальством; для упорядочения коммуникации с заказчиком.

Можно выделить следующие этапы разработки ПО:

1. Постановка задачи, которая может быть оформлена путем составления *технического задания.*

2. Собственно разработка, состоящая из:

   - анализа задач на текущем этапе;
   - проектирования решения отобранных задач;
   - кодирования и первичной отладки;
   - тестирования решения.

3. Сопровождение выпущенного ПО.

На момент написания данной книги наибольшей популярностью пользовались «agile»[121] процессы разработки. Их основной чертой является понятие *спринта* — короткой (однатри недели) итерации разработки, включающей в себя все этапы от анализа до тестирования. Каждый спринт решает конкретные задачи. В результате каждого спринта получается частично работающее решение-прототип, которое можно показать заказчику, что формирует в его глазах позитивный образ

[121] От англ. *agile* — «проворный, расторопный, подвижный».

видимого прогресса разработки и позволяет, что особенно важно, быстро поменять направление разработки, если заказчику **вдруг станет ясно**, что требуется решать иную задачу, чем виделось в начале разработки.

Некоторые характерные проблемы процесса разработки:

- *Недостаточная формализация задачи* приводит к недостаточному учету важных деталей, которые могут быть не видны в начале разработки, а в итоге приведут к необходимости полной переделки проекта.

- *Избыточная формализация и избыточное планирование* замедляют и отягощают процесс разработки и порой вынуждают решать задачи, решение которых не понадобится или не было необходимым. Не следует забывать, что идеальная модель — это сам моделируемый объект, поэтому любая рабочая модель будет иметь некие недостатки, пробелы. А программа — это обычно модель некоторой части реального или воображаемого мира. Избыточная формализация способствует гниению кода.

- *Гниение кода* [*code rot*] — рассогласование частей проекта в процессе разработки. Включает в себя два основных вида:

  - Неизменяемый и неиспользуемый (редко используемый) код постепенно перестает удовлетворять требованиям своего окружения — «гниет».

  - Слишком часто и хаотично изменяемый код легко теряет целостность, при этом нередко еще и документация не обновляется вовремя. Потеря контакта между разработчиками может привести к распаду проекта на невзаимодействующие части.

- *Подтекающие абстракции* [*leaky abstractions*[122]] — обычно не существует единственной идеальной модели рассматриваемой предметной области. В частности, не удается придумать единственную идеальную классификацию понятий. Выбранные абстракции постепенно все хуже соответствуют реальности проекта на данный момент.

- *Изменяющиеся требования* — процесс разработки длителен, а процесс коммуникации с заказчиками непредсказуем. Однажды может оказаться, что решалась не совсем та (или совсем не та) задача, которую надо было решать.

- *Низкая культура разработки* — следствие недостаточного опыта и навыков, плохой мотивации или давления жестких сроков. Высокая культура разработки предполагает:

  - *Читаемость кода.* Однажды написанный код приходится читать в среднем десять и более раз, поэтому читаемость (понятность) кода очень важна. К сожалению, восприятие кода как удобочитаемого программистом-новичком и опытным программистом сильно различается.

  - *Единообразие* — единообразные названия переменных, функций, типов. Единообразие в применяемых способах решения одних и тех же задач (в частности, не следует без жесткой необходимости использовать одновременно в одном проекте разные библиотеки, решающие одни и те же задачи).

  - *Постепенность.* Попытка решать крупные задачи сразу целиком и исправлять ошибки дописыванием

[122] «All non-trivial abstractions, to some degree, are leaky», цит. по: *Spolsky J.* The Law of Leaky Abstractions // Joel on Software : [site]. URL: `http://www.joelonsoftware.com/articles/LeakyAbstractions.html` (дата обращения: 29.07.2020).

нового кода в один прекрасный момент приведет к застопориванию работы.

Прогрессирование перечисленных патологий ведет к накоплению *технического долга* [*technical debt*] — неудачных решений, которые либо будут вызывать потери рабочего времени в будущем (в случае доработки и поддержки продукта), либо приведут к необходимости радикальной переработки с выбрасыванием большей части сделанного. В самом тяжелом случае получается «код только для записи» («write-only code») — код, который невозможно читать. Порой даже его автор не может спустя некоторое время понять, что он написал. Такой код обычно дешевле и проще выкинуть, чем разбираться в нем.

Помочь в борьбе с указанными проблемами могут, в частности, следующие простые меры:

1. *Протоколирование работы* — ведение журналов событий, вопросов и решений. Например:
   - общие события и заметки;
   - TODO: что требуется сделать в ближайшее время;
   - ISSUES: вопросы, которые следует решить, отложенные проблемы и найденные ошибки (то, что не вошло в более узкий и конкретный горизонт журнала TODO);
   - отложенная функциональность: то, что хотелось бы иметь, но решено было отложить на неопределенный срок;
   - решения: описания мер, которые позволили решить некую проблему (может пригодиться в будущем).
2. Использование некоторой современной *системы управления версиями* [*version control system, VCS*] (например, Git). Позволяет спокойно удалять код и править документацию, а также протоколировать изменения в коде (комментарии к ревизиям). При возникновении необходимости можно восстанавливать старые версии кода (и других

файлов), отслеживать конкретные изменения и находить их источник.

3. *Глоссарий*: составление словаря терминов — начальный этап анализа предметной области.

4. *Обзор кода* (решения) [*review*] — обычно выполняется перекрестно в группах разработчиков. Разработчики «меняются» результатами своей работы и проверяют друг друга (хотя бы просто читают код и указывают на непонятные или сомнительные места).

5. Анализ на основе *предпрототипа* [*pretotype*[123]] (воображаемого прототипа). Используется для выявления *прецедентов* — характерных ситуаций использования разрабатываемого продукта и желаемого функционала, а также обоснованности постановки задачи, уточнения возможной целевой аудитории решения.

6. Создание *прототипа* — максимально простой (демонстрационной) версии продукта, которая, однако, содержит все основные его части (хотя бы в виде «заглушек»). Первый прототип часто выбрасывается («напиши программу, а потом перепиши ее с нуля»). После чего делается второй прототип с тем же функционалом. При этом следует удержаться от искушения воплотить все, что не удалось воплотить в первый раз (эффект, известный как «синдром второй системы»).

7. Принцип KISS (*keep it simple and stupid*) — максимально простые решения, не нарушающие прочих принципов (в первую очередь, модульности). Нередко попытка воплотить сложное решение приводит к потере темпа и

[123] См.: *Savoia A.* What is pretotyping // Pretotyping : [site]. URL: `http://pretotyping.blogspot.ru/p/what-is-pretotyping.html` (дата обращения: 29.07.2020).

может даже повлечь крах всего проекта. Переусложнение решений («overengineering») характерно для новичков. «Как видно, совершенство достигается не тогда, когда уже нечего прибавить, но когда уже ничего нельзя отнять»[124].

8. Принцип Fail fast — как можно более раннее выявление ошибок любыми средствами (статический анализ, проверка условий в коде, многоуровневое тестирование).

14.5. Верификация

Верификация[125] — формальное обоснование (доказательство) корректности программы.

Начальный этап верификации — выявление условий, истинных на корректных параметрах или состояниях программы, чему было посвящено начало данной главы.

Элементом верификации программы можно считать проверку типов. Так как компилятор выполняет проверку типов автоматически, имеет смысл определять типы таким образом, чтобы они соответствовали выявленным инвариантам (и вообще концепциям моделируемой предметной области). Таким образом можно достигнуть выявления многих ошибок еще на этапе компиляции.

Проверку типов можно считать частным случаем *статического анализа* — выявления потенциальных ошибок в программе путем анализа кода по набору правил (в том числе, образцов типичных ошибок). Выполняется автоматически соответствующими инструментами, в частности, современными компиляторами (*предупреждения* [*warnings*]). Рекомендуется включать все виды предупреждений (ключ `-Wall` у компиляторов g++ и clang) и хотя бы иногда прогонять компиляцию

[124] А. де Сент-Экзюпери, пер. с фр. Норы Галь.
[125] От лат. *verus* — «истинный, верный».

с максимальным уровнем проверок. Выявленные места в коде следует проверить на наличие ошибки или потенциальной ошибки. Это не означает, что необходимо во что бы то ни стало избавляться от всякого кода, вызывающего предупреждения. К сожалению, возможности статического анализа ограничены и многие сообщения могут указывать на корректный код[126].

Следуя принципу fail fast, надо выявить как можно больше ошибок до запуска программы.

Во-первых, не исправленная вовремя ошибка может стоить дорого, иногда очень дорого.

Во-вторых, успешная работа программы не доказывает отсутствие ошибок.

В-третьих, тестированием невозможно доказать отсутствие ошибок в реальных программах из-за очень большого количества вариантов исполнения кода и сочетаний значений переменных.

В-четвертых, выявлять ошибки времени исполнения тяжело. Для этого требуется сначала определить точные минимальные условия возникновения ошибки, затем локализовать ее вероятное положение в коде. Последнее порой дается особенно тяжело, поскольку человек склонен искать ошибку в том месте, которое он плохо, как ему кажется, понимает. Обычно ошибка обнаруживается в другом месте — например, там, где программист не был достаточно внимателен. Еще хуже, если алгоритм реализован верно, но на деле не решает нужную задачу, так как исходит из неверных предпосылок. Корректность и адекватность любого используемого алгоритма должна быть обоснована.

[126] Вообще, такая ситуация есть недостаток системы типов, но полностью ее избежать вряд ли возможно на практике.

14.6. Протоколирование

Самым простым методом локализации ошибки исполнения является *протоколирование* [*logging*] — вывод (обычно запрограммированных вручную) отладочных сообщений (или файлов) в процессе работы программы. Результат протоколирования — *протокол работы* [*log*[127]]. В протокол заносится информация, необходимая для отслеживания корректности работы программы — например, значения некоторых переменных в определенных местах кода. Протоколирование позволяет производить отладку в ситуации отсутствия отладчика, а также может быть удобнее и эффективнее «простого» ручного тестирования. Некоторые авторы выступают за систематическое использование протоколирования вместо интерактивной отладки с помощью отладчика.

Простейшее средство протоколирования — предоставляемые стандартной библиотекой потоки вывода (cout, cerr, clog). Поток clog привязан к потоку вывода ошибок так же, как и поток cerr. Отличие заключается в том, что операции записи в cerr выполняются сразу же, в то время как запись в clog буферизуется. Поэтому частая запись в cerr сильнее сказывается на производительности программы, и clog использовать в этом плане эффективнее. Однако если исполнение программы будет прервано насильно, то данные (самые свежие данные!), записанные в буфер clog, могут не успеть быть переданы в системный поток ошибок и, таким образом, пропадут.

Пример 14.15. Протоколирование в clog

```
if (x <= y)
  clog << "\nERROR: x must be > y, x = "
       << x << ", y = " << y;
```

Недостатком использования стандартных потоков вывода является сложность управления ими, в частности, невозможность организовать надежное, не зависящее от ОС, перенаправ-

[127] Распространено использование жарг. *лог*.

ление на устройство или в произвольный файл. Стандартные потоки ввода-вывода могут быть вовсе недоступны, например, при компиляции оконного приложения.

Естественное желание сделать организацию протоколирования более удобной приводит к написанию собственных компонент или даже к использованию для этого специальных библиотек (например, Boost.Log[128]).

Пример 14.16. Минимальное протоколирование в Boost.Log

```
if (x <= y)
  BOOST_LOG_TRIVIAL(error) << "x must be > y, x = "
                           << x << ", y = " << y;
```

14.7. Тестирование

Тестирование [*testing*] — исполнение сценариев работы (*тестов*) с заранее известным желаемым поведением программы (или заранее известными результатами) и сравнение полученного и желаемого результатов. Простейший вид: ручное тестирование во время отладки.

Профилирование [*profiling*] — особый режим работы приложения, обычно реализуемый с помощью дополнительного инструмента (*профилировщика* [*profiler*]).

При профилировании выполняется некоторый фиксированный сценарий работы и осуществляется сбор статистики, такой как количество вызовов каждой функции, процент рабочего времени процессора, занятого каждой функцией или даже каждой инструкцией машинного кода (приближенно), использование оперативной памяти. Профилирование используется для оптимизации — выявления узких мест, ликвидация которых даст существенный результат.

---

[128] См.: *Semashev A.* Boost.Log v2 // Boost C++ Libraries : [site]. URL: `http://www.boost.org/libs/log/doc/html/index.html` (дата обращения: 29.07.2020).

Однако профилирование также позволяет находить скрытые ошибки (аномальное поведение), которые приводят к перерасходу ресурсов, включая утечки памяти. Впрочем, для обнаружения утечек памяти есть отдельные инструменты, например, Valgrind[129] и режим sanitize в компиляторе g++[130]. Такие инструменты отслеживают каждое обращение к памяти и способны обнаруживать ошибки обращения по неверным адресам, в частности, выходы за пределы массивов, даже если обращение идет через «посторонний» указатель, когда компилятор не «видит», к какому массиву мы пытаемся обратиться.

Впрочем, разумное применение средств современного C++ и стандартной библиотеки C++, в целом, позволяет в обычном коде гарантировать отсутствие утечек памяти и иных ресурсов. Использование системы типов, статического анализа и систематическое тестирование позволяют быстро находить и исправлять простые ошибки вроде опечаток и забытых проверок, код, вызывающий неопределенное поведение.

Модульное тестирование [*unit testing*] — написание пакетов тестов для каждого участка кода («модуля»[131]), обычно одновременно с написанием этого кода.

Пакеты тестов можно рассматривать как важное дополнение к документации: они демонстрируют инициализацию и деинициализацию ресурсов, порядок действий, показывают, как обращаться к коду, как получать и проверять результат.

Покрытие [*coverage*] — процент участков кода, для которых есть тест («покрытых тестами»). «Стопроцентное покрытие»

---

[129] См.: Valgrind : [site]. URL: `http://valgrind.org` (дата обращения: 29.07.2020).

[130] См.: Program Instrumentation Options // Using the GNU Compiler Collection (GCC) : [site]. URL: `https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html` (дата обращения: 29.07.2020).

[131] Англ. *unit* здесь можно понимать как «единица», т. е. любой в некотором смысле единый фрагмент кода.

означает, что имеется тест для каждого предполагаемого сценария использования каждой функции.

Пакет тестов [*test suite*] — отдельная группа тестов, обычно объединяющая в себе тесты одного класса (или абстрактного типа данных с его операциями) или модуля (файла исходного кода).

Тест-сборка [*test build*] — набор пакетов тестов, собираемый как особое приложение. Результатом работы такого приложения является *протокол тестирования*.

Использование специальных библиотек или IDE позволяет автоматизировать процесс тестирования: в процессе выполнения тестов автоматически создается протокол тестирования и подводится статистика (сколько удачных, сколько неудачных и т. п.).

Разработка через тестирование [*Test Driven Development*] — процесс разработки, основополагающей идеей которого является «сначала пишем тесты, затем код, который эти тесты тестируют». Во-первых, такой подход облегчает достижение высокого уровня покрытия. Во-вторых, он позволяет сначала оформить желаемые интерфейсы кода, который еще только предстоит написать, и осмыслить в процессе, как все в целом должно работать. По сути это тот же прием, что и описанный ранее предпрототип.

Регрессионное тестирование [*regressive testing*] — регулярное выполнение всех пакетов тестов. Тестируется даже отлаженный код, в который изменения не вносились. Так делается из-за того, что при изменении одной части кода могут возникнуть или проявиться ранее не найденные ошибки в другой его части. Отчасти помогает в борьбе с гниением кода.

Для выполнения регрессионного тестирования обычно организуют автоматическую сборку всего проекта и выполнение всех тестов на новой сборке по ночам («nightly builds») — утром разработчики анализируют протокол тестирования.

Для повышения результативности тестирования оно может выполняться одновременно на нескольких компьютерах с раз-

ным аппаратным и программным обеспечением (разными операционными системами, если поддерживается несколько операционных систем).

Задания для самопроверки

У Ответьте на следующие вопросы.

- Почему полезно проверять пред- и постусловия?
- В чем отличие между **static_assert** и стандартным макросом assert?
- Что называют «гниением кода»?
- Что называют «техническим долгом»?
- Что называют «предпрототипом»?
- Являются ли верификация и тестирование программы взаимоисключающими мероприятиями?

У Предположим, требуется выполнить реализацию функции **float** sqrt(**float**). Предложите набор пред- и постусловий для такой функции.

У Предположим, требуется определить функцию поиска подстроки в массиве си-строк:

```
// Возвращает успех/неуспех поиска.
// В случае успеха записывает найденную позицию
// в переменные, переданные по ссылкам.
bool find_csubstr(
    char const * const text[], size_t lines,
    char const * what,
    size_t & found_lineno, size_t & found_pos);
```

Предложите пред- и постусловия для данной функции. Попробуйте ее реализовать и протестировать (напишите для нее три-четыре теста).

Глава 15

Рекурсия

15.1. Рекурсия

Решения многих задач удобно определять рекурсивно и затем доказывать их корректность методом математической индукции.

Например, решение задачи «определить максимум набора значений $\{a_i\}_0^{n-1}$» может быть дано рекурсивной функцией max со следующим определением:

$$\max(a,n) \triangleq \begin{cases} a_0, & n=1, \\ \max\{\,\max(a,n-1), a_{n-1}\,\}, & n>1. \end{cases}$$

Как доказать (не ограничиваясь словом «очевидно»), что это определение действительно дает максимум (при условии $n>0$)? Во-первых, следует сформулировать, что́ мы в данном случае называем «максимумом». Максимум набора значений $\{a_i\}_0^{n-1}$ — это число m такое, что:

$$(\forall i)\ (0 \leqslant i < n) \Rightarrow (a_i \leqslant m),$$
$$(\exists i)\ (0 \leqslant i < n) \wedge (a_i = m).$$

Воспользуемся методом математической индукции для доказательства истинности обоих логических высказываний в

случае подстановки $m = \max(a, n)$ в соответствии с данным выше определением.

База индукции $n = 1$: для набора размера 1 имеем только один выбор $i = 0$, подстановка которого дает высказывания: $a_0 \leqslant m$, $a_0 = m$. Подставив $m = \max(a, n)$, получим $a_0 = a_0$.

Шаг индукции $n > 1$: пусть $\max(a, n-1)$ дает максимум для набора размера $n-1$ (все элементы исходного набора, кроме последнего), что означает истинность (положим $m' = \max(a, n-1)$):

$$(\forall i)\ (0 \leqslant i < n-1) \Rightarrow (a_i \leqslant m'),$$
$$(\exists i)\ (0 \leqslant i < n-1) \wedge (a_i = m').$$

По определению $m = \max(a, n) = \max\{\, a_{n-1}, m' \,\}$, откуда имеем два случая:

- $a_{n-1} \leqslant m'$, тогда $m = m'$;
- $a_{n-1} > m'$, тогда $m = a_{n-1}$.

Оба случая при подстановке m в определение максимума очевидным образом обеспечивают его истинность. Корректность рекурсивного определения доказана.

Рекурсивное определение максимума на C++ можно записать так:

```
float max(float a, float b) {
  return b < a? a: b;
}
float max(float const a[], size_t n) {
  return n == 1? a[0]: max(a[n-1], max(a, n-1));
}
```

У Как быть со случаем n==0?

Рассмотрим еще одну похожую задачу. Линейный поиск: проверить, имеется ли в массиве элемент со значением, равным заданному. В этот раз массив может быть пуст. Рекурсивное определение:

$$\mathrm{contains}(a, n, v) \triangleq \begin{cases} 0, & n = 0, \\ (a_{n-1} = v) \vee \mathrm{contains}(a, n-1, v), & n > 0. \end{cases}$$

У Докажите корректность данного выше определения.

На C++:

```
bool contains(int const a[], size_t n, int v) {
  return n == 0? false:
         a[n-1] == v || contains(a, n-1, v);
}
```

Данный пример демонстрирует то, что называется **хвостовым вызовом** [*tail call*] — функция возвращает результат вызова себя самой без выполнения каких-либо действий над ним. Это станет еще яснее видно, если переписать всю конструкцию через **if**:

```
bool contains(int const a[], size_t n, int v) {
  if (n == 0)
    return false;
  if (a[n-1] == v)
    return true;
  return contains(a, n-1, v);
}
```

Хвостовой вызов хорош тем, что его можно тривиально заменить на безусловный переход:

```
bool contains(int const a[], size_t n, int v) {
_begin:
  if (n == 0)
    return false;
  if (a[n-1] == v)
    return true;
  n = n-1;
  goto _begin;
}
```

Компиляторы способны (но не обязаны) делать данную замену при выполнении оптимизации. Безусловный переход дешевле вызова функции по затрачиваемому времени процессора. Еще более важно то, что при переходе не создается кадр стека вызовов, а значит, и нет расхода памяти стека, поэтому раскрытая в прямой переход рекурсия не может привести к переполнению стека вызовов.

Итак, хвостовой вызов воплощается циклом:

```
bool contains(int const a[], size_t n, int v) {
  while (true) {
    if (n == 0)
      return false;
    if (a[n-1] == v)
      return true;
    n = n-1;
  }
}
```

Далее, можно его несколько сократить, занеся одно из условий в условие цикла:

```
bool contains(int const a[], size_t n, int v) {
  while (n-- != 0)
    if (a[n] == v)
      return true;
  return false;
}
```

Таким образом, рекурсивные функции в форме хвостового вызова легко преобразуются в эффективные программы на основе циклов.

Рассмотрим теперь другой пример. Рекурсивная функция, вычисляющая факториал:

```
double fact(unsigned n) {
  return n == 0? 1: n*fact(n-1);
}
```

Здесь рекурсия не имеет форму хвостового вызова, поскольку после вычисления fact(n–1) требуется еще домножить его на n.

Можно ли ее привести к форме хвостового вызова?

Пусть есть функция $f(x)$, определенная рекурсивно:

$$f(x) \triangleq \begin{cases} f_0, & c(x), \\ F(f(g(x))), & \neg c(x). \end{cases}$$

Тогда можно определить функцию $f^*(x, y)$:

$$f^*(x, y) \triangleq \begin{cases} y, & c(x), \\ f^*(g(x), F(y)), & \neg c(x). \end{cases}$$

И окажется, что $f(x) \equiv f^*(x, f_0)$.

В случае факториала получаем, что $n! \equiv \mathrm{fact}(n, 1)$, где

$$\mathrm{fact}(n, f) \triangleq \begin{cases} f, & n = 0, \\ \mathrm{fact}(n-1, nf), & n \neq 0. \end{cases}$$

Или на C++:

```
double fact(unsigned n, double f = 1.) {
  return n == 0? f: fact(n-1, n*f);
}
```

Теперь мы имеем рекурсию в форме хвостового вызова и можем переписать программу в виде цикла:

```
double fact(unsigned n, double f = 1.) {
_begin:
  if (n == 0)
    return f;
  f = n*f;
  n = n-1;
  goto _begin;
}
```

или в более компактной форме (заодно избавимся от вспомогательного параметра):

```
double fact(unsigned n) {
  double f = 1.;
  while (n != 0)
    f *= n--;
  return f;
}
```

У Примените указанный прием к определенной ранее рекурсивной функции $\max(a, n)$, чтобы привести ее к форме хвостового вызова. Затем приведите ее в форму цикла **while**.

Последовательность чисел Фибоначчи традиционно определяется рекурсивно:

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}.$$

На C++:

```
double fib(unsigned n) {
  return n < 2? n: fib(n-1) + fib(n-2);
}
```

Прямое вычисление по данному определению требует времени, экспоненциально зависящего от n. Убедиться в этом можно, например, попробовав вычислить fib(40), fib(45), fib(50). При этом, если идти по определению «снизу вверх» до n, то, очевидно, понадобится лишь $n-1$ операций сложения для получения результата (линейное время). Из этого можно сделать вывод, что такую конструкцию можно привести к форме хвостового вызова.

Применив тот же прием, что был применен к функции вычисления факториала, мы можем получить что-то такое:

```
double fib(unsigned n, double y=1.) {
  return n < 2? y: fib(n-1, y + fib(n-2));
}
```

Здесь хвостовым стал только один вызов, так что линейности мы еще не достигли. Но важнее то, что нам пришлось изменить семантику функции: теперь fib(0)==1. Прежде чем продолжить, обратим внимание на то, что последовательности типа «числа Фибоначчи» образуют двухпараметрическое семейство, если положить:

$$F_0^{a,b} = a, \quad F_1^{a,b} = b, \quad F_n^{a,b} = F_{n-1}^{a,b} + F_{n-2}^{a,b}.$$

Эти параметры a и b должны быть переданы *оба*. Далее мы должны проверить два особых случая: $n = 0$ (тогда вернем a) и $n = 1$ (тогда вернем b). Если это сделать, то мы получим желаемое:

```
double fib(unsigned n, double a=0., double b=1.) {
  if (n == 0) return a;
  if (n == 1) return b;
  return fib(n-1, b, a + b);
}
```

Теперь у нас рекурсия в форме хвостового вызова, и ее легко раскрыть в цикл. Немного доработав код, получим:

```
double fib(unsigned n, double a=0., double b=1.) {
  if (n == 0) return a;
  while (n-- != 1) {
    auto const s = a + b;
    a = b;
    b = s;
  }
  return b;
}
```

15.2. Префиксный калькулятор

Для описания синтаксиса той или иной формы записи (формального языка) можно использовать конструкцию, известную как *порождающая грамматика* [*generative grammar*]. Порождающая грамматика описывает процесс порождения строк языка как последовательность применения правил замены подстрок.

Порождающая грамматика — четверка (T, N, R, S), где T — непустое множество *терминальных символов* (*терминалов*), N — непустое множество *нетерминальных символов* (*нетерминалов*), R — множество правил замены вида «левая часть $\rightarrow$ правая часть», S — начальный символ. При этом должны выполняться правила: $T \cap N = \varnothing$, $S \in N$, левая часть каждого правила из R содержит хотя бы один нетерминал из N.

Пример. Пусть задана грамматика следующего вида:

- $T = \{(,)\}$;
- $N = \{S\}$;
- $R = \{S \rightarrow SS, S \rightarrow (S), S \rightarrow \epsilon\}$, где ϵ — пустая строка.

Данная грамматика описывает всевозможные корректные *скобочные формы*, например, (), ()(), (()), ((()))(()) и т. д. (*язык скобочных форм*). Правила применяются до тех пор, пока это возможно (строка содержит подстроки, совпадающие с левой частью какого-либо из правил).

Иерархия грамматик Хомского[132] — четыре вложенных типа грамматик (каждый следующий тип является строгим подмножеством предыдущего):

- тип 0: *произвольные грамматики*, нет ограничений на вид правил. Распознаватель — универсальный компьютер, например, машина Тьюринга;

- тип 1: *контекстно-зависимые грамматики* [*context-sensitive grammars*], все правила имеют вид: $\alpha A \beta \to \alpha \gamma \beta$, где $A \in N$, а α, β, γ — произвольные последовательности терминалов и нетерминалов (элементы $(T \cup N)^*$). Распознаватель — *линейно-ограниченный автомат* [*linearly bound automaton, LBA*], т. е. универсальный компьютер с объемом памяти, ограниченным сверху линейной функцией длины ввода;

- тип 2: *контекстно-свободные грамматики* [*context-free grammars*], все правила имеют вид: $A \to \alpha$, где $A \in N$, $\alpha \in (T \cup N)^*$. Распознаватель — *автомат с магазинной памятью, АМП* [*pushdown automaton, PDA*];

- тип 3: *регулярные грамматики* [*regular grammars*], все правила имеют вид $A \to a$ и либо $A \to aB$, либо $A \to Ba$, где $A \in N$, $B \in N$ (A и B могут совпадать), $a \in T$. Распознаватель — *конечный автомат* [*finite automaton*], т. е. автомат с конечным числом состояний.

Конечные автоматы рассмотрены в следующей главе. Автоматы с магазинной памятью можно представлять в виде конечных автоматов, дополненных неограниченным *стеком* («магазином»).

На практике основной интерес представляют грамматики типов 2 и 3. Они, в частности, используются для определения большинства языков программирования и форматов файлов.

---

[132] Н. Хомский (N. Chomsky, род. 1928) — американский лингвист и публицист.

Подробности, касающиеся порождающих грамматик и связанного с ними математического аппарата и теоретических результатов, составляют весьма обширную тему и выходят за рамки данной книги.

Рассмотрим следующую задачу: дано выражение в *префиксной форме*[133], требуется вычислить его значение.

Префиксная форма — запись выражения, в которой операция всегда предшествует своим операндам. Для простоты будем считать, что все операции имеют два операнда, а элементарными операндами являются числа.

Примеры:

- `+ 1 2` равно 3.
- `+++ 1 2 3 4` равно 10.
- `*+ 2 2 3` равно 12.
- `+ - 3 1 * 3 4` равно 14.

Грамматика префиксной формы дает нам ее строгое определение. Это контекстно-свободная грамматика.

Терминалы: знаки операций и знаки записи чисел.

Нетерминалы: Число, Выражение, Операция.

Правила (для простоты опускаем определение числа):

- Выражение → Число
  - значение равно этому числу;
- Выражение → Операция Выражение Выражение
  - значение равно результату операции, операндами которой являются стоящие за ней выражения.

[133] Известной также как *польская запись* [*Polish notation*].

Распознавание построено на сопоставлении правых частей правил с текстом и «свертке» распознанных частей с заменой правых частей правил на левые части. Здесь правая часть каждого правила начинается с символа, который можно распознать сразу (число или операция).

Для распознавания можно написать программу, в которой каждому нетерминалу соответствует функция, читающая символы из потока ввода и «принимающая» или «не принимающая» их как запись этого нетерминала по правилам, его задающим. Данный метод называется *рекурсивным спуском* [*recursive descent*]. В данном случае определим только две функции: number (число) и prefix (выражение).

```
double number() {
  if (double val; cin >> val)
    return val;
  return NAN;
}
```

Ради простоты игнорируем ошибки (в случае ошибки возвращаем нечисло).

Функция peek возвращает следующий непробельный символ в cin, не удаляя его из потока.

```
char peek() {
  char ch = 0;
  if (cin >> ch)
    cin.unget();
  return ch;
}

double prefix() {
  // Распознать операцию.
  enum { Add, Sub, Mul, Div, Pow } op;
  switch (peek()) {
  case '+': op = Add; break;
  case '-': op = Sub; break;
  case '*': op = Mul; break;
  case '/': op = Div; break;
  case '^': op = Pow; break;
```

```
  default: // не операция, а число
    return number();
  }

  cin.ignore(); // убрать знак операции
  // Вычислить операнды.
  double const
    x = prefix(),
    y = prefix();

  // Вычислить операцию.
  switch (op) {
  case Add: return x + y;
  case Sub: return x - y;
  case Mul: return x * y;
  case Div: return x / y;
  case Pow: return pow(x, y);
  default: // невозможно
    return NAN;
  }
}
```

Теперь рассмотрим более сложный пример. Вернемся к программе CheMass из предыдущей главы. Применим к этой задаче метод рекурсивного спуска. Запишем грамматику формул:

- Формула → *пусто*
  - формула может быть пустой, тогда считаем массу нулевой;
- Формула → СкФ Индекс Формула
  - слева отделяется скобочная форма;
- Формула → Элемент Индекс Формула
  - слева отделяется элемент;
- СкФ → ‘(’ Формула ‘)’

– скобочная форма — это формула в скобках;

- Элемент → ЗаглавнаяБуква
  - название элемента из одной буквы. Например, H, P или C;
- Элемент → ЗаглавнаяБуква СтрочнаяБуква
  - название элемента из двух букв (естественно, без пробела между ними). Например, Na, Be или Se;
- Индекс → *пусто*
  - если индекс не указан, то считаем его равным единице;
- Индекс → ЦелоеЧисло
  - множитель массы элемента или скобочной формы.

Таким образом, формула может начинаться либо на заглавную букву, либо на открывающую скобку.

Предполагается, что у нас есть два массива: имена элементов и массы элементов.

```
// Максимально возможное число элементов.
int const MAX_ELEMENTS = 200;
// Сколько всего элементов.
int elements;
// Массив названий элементов.
string name[MAX_ELEMENTS];
// Массив масс элементов.
double mass[MAX_ELEMENTS];
```

Теперь код распознавателя. Предполагается, что у нас есть функция peek, использовавшаяся в предыдущем примере.

```
// Правило грамматики «Формула».
double formula() {
  // Объявления процедур.
  double element();
```

```
  double skf();
  int index();
  // Выбор продукции.
  char const next = peek();
  if (next > 0) {
    double first = 0;
    if (isupper(next) && (first = element())) {
      int coeff = index();
      return first*coeff + formula();
    }
    if (next == '(' && (first = skf()))
      return first + formula();
  }
  return 0;
}

// Правило грамматики «Элемент».
double element() {
  char el[3] { peek() }; // имя элемента
  if (!isupper(el[0])) {
    clog << "ERR: element can't begin with '"
         << el[0] << "'\n";
    return 0;
  }

  cin.ignore();
  el[1] = peek();
  if (islower(el[1]))
    cin.ignore();
  else
    el[1] = '\0';

  // Найти имя среди элементов массива name.
  for (int i = 0; i < elements; ++i)
    if (name[i] == el)
      return mass[i]; // нашли
  // Не нашли.
  clog << "ERR: unknown element: " << el << '\n';
  return 0;
}
```

```
// Правило грамматики «Скф» (скобочная форма).
double skf() {
  // Объявления процедур.
  int index();
  if (peek() != '(') {
    clog << "ERR: internal error: skf expects (\n";
    return 0;
  }

  cin.ignore(); // убрать (
  double result = formula();
  if (peek() != ')')
    clog << "ERR: expected )\n";
  else
    cin.ignore(); // убрать )
  return result*index();
}

// Правило грамматики «Индекс».
int index() {
  int value = 1;
  if (isdigit(peek()))
    cin >> value;
  return value;
}
```

15.3. Постфиксный калькулятор

Вернемся к калькулятору и поменяем грамматику на следующую: Правила:

- Выражение → Число
  - значение равно этому числу;
- Выражение → Выражение Выражение Операция
  - значение равно результату операции, примененной к выражениям, стоящим перед ней (два операнда).

Данная грамматика соответствует *постфиксной форме*[134] представления выражений.

Если мы попробуем сразу написать рекурсивный разбор аналогично тому, как это было сделано для префиксной формы, то получим бесконечную рекурсию, поскольку выражение должно сначала вычислить операнд — снова выражение.

Это, впрочем, не означает, что записать аналог невозможно. Приведем сразу готовый код:

```
double postfix(double x = 0, double y = 0) {
  enum { Add, Sub, Mul, Div, Pow } op;
  switch (peek()) {
  case '+': op = Add; break;
  case '-': op = Sub; break;
  case '*': op = Mul; break;
  case '/': op = Div; break;
  case '^': op = Pow; break;
  default:
    if (double z; cin >> z)
      return postfix(x, postfix(y, z));
    return y;
  }

  cin.ignore();
  switch (op) {
  case Add: return x + y;
  case Sub: return x - y;
  case Mul: return x * y;
  case Div: return x / y;
  case Pow: return pow(x, y);
  default: // ошибка в коде
    assert(!"Unknown operation.");
    return y;
  }
}
```

[134] Известной также как *обратная польская запись* [*reverse Polish notation, RPN*].

Функция postfix сразу «знает» свои операнды (x и y). Данный прием отчасти напоминает приведение к форме хвостового вызова рекурсивного определения функции, вычисляющей числа Фибоначчи.

Один рекурсивный вызов можно убрать, заменив циклом:

```
double postfix(double x = 0, double y = 0) {
  enum { Add, Sub, Mul, Div, Pow } op;
  for (;;) {
    switch (peek()) {
    case '+': op = Add; break;
    case '-': op = Sub; break;
    case '*': op = Mul; break;
    case '/': op = Div; break;
    case '^': op = Pow; break;
    default:
      if (double z; cin >> z) {
        y = postfix(y, z);
        continue;
      }
      return y;
    }

    cin.ignore();
    switch (op) {
    case Add: return x + y;
    case Sub: return x - y;
    case Mul: return x * y;
    case Div: return x / y;
    case Pow: return pow(x, y);
    default: // ошибка в коде
      assert(!"Unknown operation.");
      return y;
    }
  }
}
```

Однако здесь у нас не получится сделать хвостовую рекурсию с ограниченным промежуточным состоянием (фиксированным объемом памяти), поскольку грамматика не является

регулярной и на каждом шаге n возможно порядка $(1+p)^n$ разных комбинаций *операция-число*, где p — число возможных операций.

Контекстно-свободные грамматики могут быть распознаны автоматом с магазинной памятью, т. е. со *стеком*, а значит, избавиться от рекурсии мы можем, используя явный стек. В конце концов, рекурсия организована на основе стека — стека вызовов, поэтому ее всегда можно убрать, заменив на цикл плюс явный стек промежуточных состояний.

Стек [*stack*] — структура данных, позволяющая добавлять и удалять элементы по одному, при этом элементы удаляются в порядке, обратном добавлению («последним зашел — первым вышел»), и возможен доступ к последнему добавленному элементу (*вершине стека* [*stack top*]).

Здесь для простоты будем использовать стандартный стек, доступный при включении заголовочного файла **stack**. Объекты std::stack предоставляют четыре основных действия:

- s.empty() — возвращает истинность условия «стек s пуст»;
- s.push(item) — положить на стек s новый элемент item;
- s.pop() — убрать из стека s вершину стека (предусловие: стек не пуст, иначе неопределенное поведение);
- s.top() — получить ссылку на элемент-вершину стека s (предусловие: стек не пуст, иначе неопределенное поведение).

```
double postfix(double x = 0., double y = 0.)
{
  stack<double> args;
  enum { Add, Sub, Mul, Div, Pow } op;
  for (;;) {
    switch (peek()) {
    case '+': op = Add; break;
    case '-': op = Sub; break;
    case '*': op = Mul; break;
```

```
      case '/': op = Div; break;
      case '^': op = Pow; break;
      default:
        if (double z; cin >> z) {
          args.push(x); // поместить в стек
          x = y;
          y = z;
          continue;
        }
        return y;
      }

      cin.ignore();
      switch (op) {
      case Add: x += y; break;
      case Sub: x -= y; break;
      case Mul: x *= y; break;
      case Div: x /= y; break;
      case Pow: x = pow(x, y); break;
      default: // ошибка в коде
        assert(!"Unknown operation.");
        return y;
      }

      y = x;
      x = NAN;
      if (!args.empty()) {
        x = args.top(); // извлечь из стека
        args.pop();
      }
    }
}
```

Естественно, можно избавиться от переменных x и y, используя вместо них два верхних элемента на стеке (y — вершина стека, x — элемент, который лежит под вершиной).

15.4. Инфиксный калькулятор

Теперь, наконец, перейдем к «обычной» форме записи выражений — *инфиксной*, в которой операции записываются между операндами, а для управления порядком связывания можно использовать круглые скобки.

Запишем упрощенный вариант грамматики (без приоритетов и унарных операций), подготовленный для применения метода рекурсивного спуска. Правила:

- Выражение → Терм Продолжение
  - значение выражения равно значению терма (*левая часть*), к которому применили продолжение (если оно есть — оно может быть пустой строкой, тогда остается только значение терма);
- Продолжение → *пусто*
  - случай, когда продолжения нет;
- Продолжение → Операция Выражение
  - случай, когда продолжение есть. Значение — (*левая часть*) операция (*правая часть*);
- Терм → Число
  - значение равно значению числа;
- Терм → '(' Выражение ')'
  - значение равно значению скобочной формы;

Определим функции, распознающие выражение и терм:

```
double infix() { // выражение
  double term();
  double continuation(double);
  return continuation(term());
}
```

```
double term() {
  double x = NAN;
  if (peek() == '(') { // скобочная форма
    cin.ignore();
    x = infix();
    if (peek() != ')')
      clog << "Expected␣).\n";
    else
      cin.ignore();
  }
  else if (cin >> x; cin.fail()) // число
    clog << "Expected␣(␣or␣number.\n";
  return x;
}
```

Основную нагрузку несет функция continuation (продолжение). Для выполнения операции продолжение должно знать левую часть, поэтому передадим ее параметром. Наконец, рекурсивное определение продолжения очень легко привести к циклическому виду (последовательность термов, разделенных операциями), что мы и сделаем сразу. В итоге, получим следующее определение:

```
double continuation(double x) {
  enum { Add, Sub, Mul, Div, Pow } op;
  for (;;) {
    switch (peek()) {
    case '+': op = Add; break;
    case '-': op = Sub; break;
    case '*': op = Mul; break;
    case '/': op = Div; break;
    case '^': op = Pow; break;
    default:
      return x;
    }

    cin.ignore();
    double y = term();
    switch (op) {
```

```
      case Add: x += y; break;
      case Sub: x -= y; break;
      case Mul: x *= y; break;
      case Div: x /= y; break;
      case Pow: x = pow(x, y); break;
      default: // невозможно, ошибка в коде
        assert(!"Internal error: unknown operation.");
        return y;
      }
    }
}
```

У Приведите пример корректного инфиксного выражения, не предусмотренного записанной выше грамматикой (естественно, не используя какие-либо операции кроме пяти предусмотренных).

Мы не будем приводить этот код к форме, использующей явный стек. Вместо этого опишем алгоритм, позволяющий вычислить значение выражения, заданного в инфиксной форме с заданными приоритетами и порядками связывания (слева-направо или справа-налево) операций.

Данный алгоритм был предложен Э. Дейкстрой[135] и носит название «алгоритм сортировочной станции» [*shunting yard algorithm*]. Вместо стека операндов он использует стек операций. Для вычисления выражения нам понадобится стек операндов (по сути, алгоритм переводит инфиксную запись в постфиксную). Здесь рассмотрен упрощенный вариант алгоритма.

Итак, у нас есть стек операций, в который также можно класть открывающие скобки. У нас есть интерпретатор постфиксной формы, которому мы «отправляем» числа и операции. Алгоритм заключается в том, чтобы считывать с потока ввода лексемы, пока это возможно, выполняя для каждой лексемы tok следующее:

- если tok = '(', положить ее в стек операций;

[135] См.: *Dijkstra E. W.* Algol 60 translation: An algol 60 translator for the x1 and making a translator for algol 60. Amsterdam, 1961.

- если tok — число, отправить его;
- если tok = ‘)’, отправить все операции из стека операций до ‘(’, убрать ‘(’ из стека;
- если tok — операция, то отправлять операции из стека до тех пор, пока:
  - он не опустеет,
  - или не встретится ‘(’,
  - или не встретится операция с меньшим приоритетом, чем у tok,
  - или не встретится операция с приоритетом, равным приоритету tok (если tok связывается справа налево).

В конце отправить все операции из стека.

Приоритет и порядок связывания будем определять с помощью функций prec и right:

```
int prec(char op) {
  switch (op) {
  case '+': case '-': return 100;
  case '*': case '/': return 200;
  case '^': return 300;
  default: return 0; // неизвестная операция
  }
}

// Правоассоциирующая операция?
bool right(char op) {
  return op == '^';
}
```

Так как мы хотим сразу вычислять значение выражения, то удобно организовать разделяемые данные (стеки) в виде полей объекта и разбить алгоритм на набор процедур.

```
struct Shyard {
public:
  double operator()() {
    for (double x; cin;) {
      switch (auto op = peek()) {
      case '(':
        open();
        cin.ignore();
        break;
      case '+':case '-':case '*':case '/':case '^':
        operation(op);
        cin.ignore();
        break;
      case ')':
        close();
        cin.ignore();
        break;
      default: // число
        if (cin >> x)
          args.push(x);
      }
    }
    finish();
    return args.empty()? NAN: args.top();
  }

private:
  stack<char> ops;
  stack<double> args;
  void open() { ops.push('('); }
  void close() {
    for (;;) {
      if (ops.empty()) {
        clog << "Unbalanced␣).\n";
        break;
      }
      if (ops.top() == '(') {
        ops.pop();
        break;
      }
```

```
    do_top();
  }
}

void operation(char op) {
  while (!ops.empty()) {
    if (ops.top() == '(')
      break;
    auto p1 = prec(ops.top()), p2 = prec(op);
    if (p1 < p2 || (right(op) && p1 == p2))
      break;
    do_top();
  }
  ops.push(op);
}

void finish() {
  while (!ops.empty()) {
    if (ops.top() == '(') {
      clog << "Unbalanced (.\n";
      break;
    }
    do_top();
  }
}

// Выполнить операцию на вершине стека.
void do_top() {
  if (args.empty()) {
    clog << "Not enough arguments for "
         << ops.top() << ".\n";
    ops.pop();
    return;
  }

  double y = args.top();
  args.pop();

  if (args.empty()) {
    clog << "Not enough arguments for "
```

```
            << ops.top() << ".\n";
        ops.pop();
        return;
      }

      double x = args.top();
      switch (ops.top()) {
      case '+': x += y; break;
      case '-': x -= y; break;
      case '*': x *= y; break;
      case '/': x /= y; break;
      case '^': x = pow(x, y); break;
      default:
        clog << "Invalid operation: "
             << ops.top() << "\n";
      }

      args.top() = x;
      ops.pop();
    }
};
```

Задания для самопроверки

У Ответьте на следующие вопросы.

- Что общего между рекурсивным определением последовательности и доказательством с помощью математической индукции?
- Чем хвостовой вызов лучше обычного?
- Какие основные операции предлагает стек?
- К какому типу грамматик в иерархии Хомского относится приведенная в главе грамматика скобочных форм?
- В чем преимущества и недостатки префиксной или постфиксной формы записи выражений по сравнению с инфиксной формой?

- Почему для префиксной формы записи выражений невозможно дать регулярную грамматику?

У Пусть дано рекурсивное определение последовательности вида:

$$H_i = \begin{cases} a, & i = 0, \\ b, & i = 1, \\ c, & i = 2, \\ H_{i-3} + H_{i-2} + H_{i-1}, & i > 2. \end{cases}$$

Напишите функцию, рекурсивно вычисляющую H_i. Переделайте ее таким образом, чтобы она содержала единственный хвостовой вызов самой себя. Перепишите хвостовой вызов в виде цикла.

У Попробуйте добавить поддержку унарных операций в программу, реализующую алгоритм сортировочной станции.

Глава 16

Конечные автоматы

16.1. Определения

Конечный автомат [*finite automaton*[136], *finite state machine*] — автомат, который:

- работает в пошаговом режиме;
- на каждом шаге может принимать одно состояние из конечного набора возможных состояний;
- начав с некоторого (начального) состояния, на каждом шаге считывает один символ из строки ввода;
- считав символ, переходит в новое состояние в зависимости от текущего состояния и считанного символа, следуя некоторому наперед заданному правилу. При этом возможно выполнение некоторого действия;
- если для заданной комбинации состояния и входного символа правило перехода не определено, то говорят, что автомат **застопорился** [*got stuck*], что является частным случаем останова (как, например, при завершении ввода).

[136] Во мн. ч. *automata*.

Различают два основных типа конечных автоматов:

- *автоматы-классификаторы* [*classifier automata*] определяют принадлежность ввода одному из конечного набора заранее заданных множеств. Важный частный случай — *автоматы-распознаватели* [*recognizer automata, acceptor automata*], вычисляющие предикат, т. е. проверяющие принадлежность строки ввода некоторому формальному языку[137];

- *автоматы-преобразователи* [*transducer automata*] (также называемые *конечные автоматы с выходом*) преобразуют строку ввода в строку вывода.

Область применения конечных автоматов включает в себя: анализ корректности ввода и программирование трансляторов (синтаксический анализ, «регулярные выражения»), создание систем управления (аппаратные контроллеры, сетевые протоколы), моделирование систем объектов, включая физические устройства, а также лингвистические модели (в частности, порождение регулярных форм слов по набору правил).

Математически конечный автомат-распознаватель задается пятеркой $(\mathcal{A}, \Sigma, s_0, a, \rho)$, где:

- $\mathcal{A}$ — алфавит символов ввода;

- Σ — непустое конечное множество возможных состояний автомата;

- $s_0 \in \Sigma$ — начальное состояние;

- $a\colon \Sigma \mapsto \{\,0, 1\,\}$ — предикат, определяющий, является ли заданное конечное состояние **принимающим** [*accepting*];

[137] В теории автоматов теорема Клини устанавливает связь между конечными автоматами и *регулярными языками* — классом языков, которые могут распознавать конечные автоматы. Для простоты можно считать, что автомат задает распознаваемый им язык, хотя один и тот же язык могут распознавать разные автоматы.

- $\rho\colon \mathcal{A} \times \Sigma \mapsto \Sigma$ — правила перехода, на каждом шаге определяющие следующее состояние автомата по текущему входному символу и предыдущему состоянию. Обычно перечисляется лишь часть всех возможных исходных комбинаций пар символ-состояние. В этом случае предполагается, что если случается неуказанная комбинация, то автомат останавливается (застопоривается и не принимает ввод).

С практической точки зрения ρ бывает удобно задавать не для всех пар из $\mathcal{A} \times \Sigma$. Можно считать, что если переход для некоторой комбинации не определен, то автомат переходит в особое состояние — «застопорился», из которого нет выхода — все переходы из него возможны только в него же. Естественно, это состояние не является принимающим.

Если в конце ввода автомат оказывается в принимающем состоянии, то строка ввода распознана как принадлежащая языку, иначе — как не принадлежащая.

Конечный автомат-преобразователь задается шестеркой $(\mathcal{A}, \mathcal{B}, \Sigma, s_0, a, \rho)$, где $\mathcal{B}$ есть алфавит вывода, а функция правил перехода ρ возвращает пары из $\Sigma \times \mathcal{B}$ или даже $\Sigma \times \mathcal{B}^*$ (в ответ на какой-то переход можно выдавать на вывод целую заранее заданную строку или ничего не выдавать — в случае пустой строки).

Описанный выше автомат-преобразователь также называется *автоматом Мили* [*Mealy machine*]. В автоматах Мили своя выходная строка отождествляется с каждым переходом.

Существует более узкий класс автоматов, называемых *автоматами Мура* [*Moore machine*], в которых выход формируется только в зависимости от состояния, в которое приходит автомат. Можно считать, что в автомате Мура отображение ρ имеет вид $\mathcal{A} \times \Sigma \mapsto \Sigma$, и задано дополнительное отображение $\beta\colon \Sigma \mapsto \mathcal{B}^*$. При каждом переходе автомат выводит β от нового состояния. Таким образом, в автомате Мура выходная строка отождествляется с каждым состоянием.

Следует понимать, что «преобразование входного текста в выходной» есть абстрактная формулировка. Элементами $\mathcal{B}$ могут быть некие события или действия, тогда $\mathcal{B}^*$ — множество всех возможных последовательностей событий или применений действий из множества $\mathcal{B}$. Такой автомат может быть управляющим устройством, посылающим сигналы некой управляемой системе.

Наконец, автомат-распознаватель («автомат без выхода») можно считать частным случаем автомата с выходом, в котором (в простейшем случае)

$$\mathcal{B} = \{\text{ ожидает}, \text{застопорился}, \text{принял }\}$$

— набор событий, которые могут происходить при распознавании строки.

Часто конечные автоматы изображаются в виде диаграмм, на которых состояния изображены в виде кружков или овалов, которые соединены обозначающими переходы стрелками, помеченными соответствующими входными символами. Непомеченные стрелки отвечают переходу «при всех прочих (неуказанных) входных символах». Принимающие состояния помечают двойной или более широкой границей кружка, а входное состояние стрелкой, не имеющей привязки к состоянию исхода.

На диаграмме мы можем изобразить сразу несколько стрелок с одинаковым условием перехода, исходящих из одного состояния. Или даже вообще без условия перехода. Как ни странно, изображение автомата такого рода вовсе не обязательно считать ошибочным. Если предположить, что *автомат переходит по всем подходящим стрелкам сразу*, то мы получим то, что называется **недетерминированный конечный автомат** (НКА) [*non-deterministic finite automaton, NFA*].

Если же потребовать, чтобы все стрелки имели условие и чтобы все условия стрелок, исходящих из одного состояния, были взаимоисключающими, то автомат будет называться **детерминированным** (ДКА) — в каждой ситуации будет не более одного возможного перехода.

Недетерминированный конечный автомат-классификатор всегда можно заменить эквивалентным детерминированным конечным автоматом, множество состояний которого есть подмножество булеана множества состояний недетерминированного конечного автомата (а это конечное множество). В данной книге мы ограничимся использованием ДКА.

Диаграммы удобны при составлении и анализе конечных автоматов. Построенную диаграмму легко перевести в код на языке программирования, моделирующий (реализующий) соответствующий ДКА. Однако сделать это можно различными способами. Выбор конкретного способа зависит от:

1. Удобства интеграции во внешний программный код.
2. Простоты внесения изменений в полученный код (изменения поведения автомата).
3. Требований к быстродействию.

С точки зрения п. 1 нужно определить:

- как автомат будет получать входные символы: будет ли его вызывать внешний код, передавая каждый следующий символ, или же автомат, будучи запущен, будет читать символы самостоятельно из некоторого источника (потока, строки, массива и т. п.);
- должен ли автомат явно представлять состояние, в котором находится (есть ли необходимость его передавать вовне или сохранять для последующего восстановления), и в какой форме его следует представлять;
- есть ли необходимость явно сигнализировать застопоривание или принятие строки или вообще любое изменение состояния;
- есть ли необходимость изменять набор правил во время исполнения программного кода, или же они могут быть зафиксированы в программе.

Существуют библиотеки, позволяющие строить произвольные конечные автоматы и решающие все или многие из упомянутых вопросов. Здесь же мы ограничимся «ручным» кодированием достаточно простых случаев.

16.2. Пример простого ДКА

Пусть дано:

- $\mathcal{A} = \{ a, b, c \}$ (три возможных входных символа);
- $\Sigma = \{ 0, 1, 2, 3, 4 \}$ (пять возможных состояний автомата);
- $s_0 = 0$;
- $a(s) = [s = 4]$ (одно принимающее состояние);
- набор правил перехода показан на рис. 16.1.

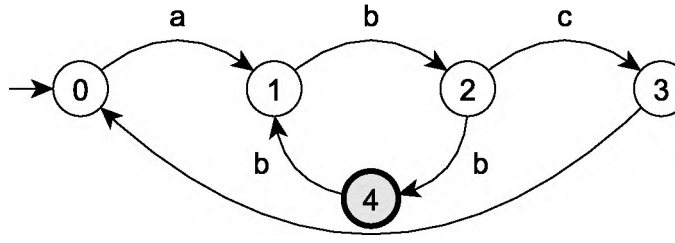


Рис. 16.1. Пример конечного автомата

Какие строки принимает этот автомат?

- Последовательность состояний 0-1-2-3-0-... соответствует пропуску любого количества повторений $abcs$, где s — любой символ. Если же после этого встретится последовательность abb, то такая строка будет принята.

- То же самое можно сказать об $abbbb$ (последовательность состояний 0-1-2-4-1-2-4). Вообще, благодаря циклу 1-2-4 автомат принимает «хвост» вида $abb(bbb)*$. Звездочкой здесь обозначено произвольное число повторений последовательности в скобках, т. е. a и $(3i-1)$ подряд идущих b, $i \in \mathbb{N}$.

- Благодаря этому же циклу автомат может пропускать строчки, составленные не только из $abcs$, но из $ab(bbb)*cs$, т. е. с группами по $(3i-2)$ подряд идущих символов b.

Закодируем рассмотренный выше автомат в классической форме «цикл+**switch** по состоянию». Состояние в этом случае хранится в специально отведенной для этого переменной.

Цикл будет считывать символы с потока ввода, пока это возможно, или не произойдет застопоривание. Признак отсутствия застопоривания будем хранить в явной форме в переменной running. Результат функции — ответ на вопрос «пришел ли в конце работы автомат в принимающее состояние».

Пример 16.1. Пример реализации конечного автомата

```
bool dfa_example(std::istream & is) {
  int state = 0;
  bool running = true;
  for (char input; is.get(input) && running;)
  switch (state) {
  case 0:
    if (input == 'a')
      state = 1;
    else
      running = false;
    break;
  case 1:
    if (input == 'b')
      state = 2;
    else
      running = false;
    break;
```

```
    case 2:
      if (input == 'c')
        state = 3;
      else if (input == 'b')
        state = 4;
      else
        running = false;
      break;
    case 3:
      state = 0;
      break;
    case 4:
      if (input == 'b')
        state = 1;
      else
        running = false;
      break;
    }

    // Состояние 4 — принимающее.
    return state == 4;
}
```

Тот же автомат можно представить в объектной форме. Объект автомата будет обрабатывать по одному символу, который передается ему извне. Таким образом, в данном примере цикл **for** вынесен в код, использующий автомат.

Пример 16.2. Конечный автомат-объект

```
class Dfa_example {
  int state = 0;
public:
  // Проверить, находится ли автомат
  // в принимающем состоянии.
  bool accepts() const {
    return state == 4;
  }
  // Возвращает ложь, если автомат застопорился.
  // Возвращает истину, если автомат не застопорился
  //  и ожидает следующий символ.
```

```
  bool operator()(char input) {
    switch (state) {
    case 0:
      if (input == 'a')
        state = 1;
      else
        return false;
      break;
    case 1:
      if (input == 'b')
        state = 2;
      else
        return false;
      break;
    case 2:
      if (input == 'c')
        state = 3;
      else if (input == 'b')
        state = 4;
      else
        return false;
      break;
    case 3:
      state = 0;
      break;
    case 4:
      if (input == 'b')
        state = 1;
      else
        return false;
      break;
    }

    return true; // нет застопоривания
  }
};
```

16.3. Символьные константы

Рассмотрим задачу распознавания символьной константы C++. Определим упрощенный синтаксис символьной константы: пусть это будет непустая последовательность символов, заключенная в апострофы '. Возможно экранирование апострофа символом \.

Возьмем следующую диаграмму автомата-распознавателя, показанную на рис. 16.2:

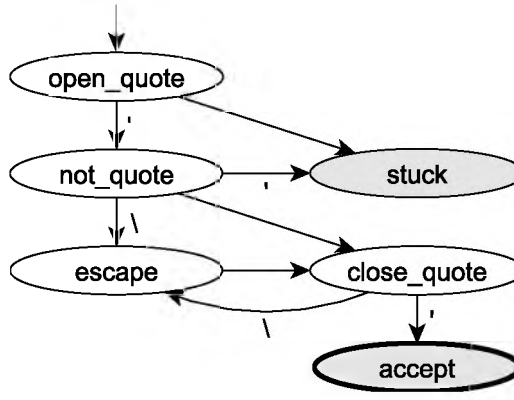


Рис. 16.2. Распознаватель записи символьной константы

Данную диаграмму легко «перевести» на C++. В качестве ввода будем использовать диапазон массива символов, заданный указателем на начало и указателем на символ, стоящий после последнего («полуинтервал»).

Пример 16.3. Символьный литерал v.1

```
// Просматривает диапазон массива символов [from, to).
// Возвращает указатель на символ,
// следующий за последним принятым (т. е. за константой).
// Возвращает nullptr, если литерал не был распознан.
char const * char_lit(
```

```
    char const * from,
    char const * to) {
  // Возможные состояния автомата:
  enum {
    open_quote,   // ожидается открывающая '
    close_quote,  // ожидается закрывающая '
    not_quote,    // первый символ литерала, не '
    escape,       // после \
    accepted,     // литерал закончился
    stuck         // застопоривание: не литерал
  } state = open_quote;
  while (from != to) {
    char const ch = *from++;
    switch (state) {
    case open_quote:
      state = ch == '\''? not_quote : stuck;
      break;
    case not_quote:
      switch (ch) {
      case '\\': state = escape; break;
      case '\'': state = stuck; break;
      default: state = close_quote;
      }
      break;
    case close_quote:
      switch (ch) {
      case '\\': state = escape; break;
      case '\'': state = accepted; return from;
      default:; /* ничего не менять */
      }
      break;
    case escape: // просто пропустить символ
      state = close_quote;
      break;
    case stuck:
      return nullptr;
    default:
      assert(!"impossible FSM state occured");
    }
  }
```

```
  return nullptr; // слишком рано кончилась строка
}
```

Не составит труда заметить, что полученный код несколько избыточен. Мы можем убрать некоторые состояния автомата: достаточно проверить, что первый символ — кавычка до входа в цикл, и тогда состояние open_quote не нужно. Состояния stuck и accepted можно убрать, потому что переход в них соответствует возвращению значения из функции.

Пример 16.4. Символьный литерал v.2

```
char const * char_lit(
    char const * from,
    char const * to) {
  // Вытащим open_quote: данное состояние
  //  имеет смысл только на первом символе.
  if (from == to || *from++ != '\'')
    return nullptr;
  // Возможные состояния автомата:
  enum {
    close_quote, // ожидается закрывающая '
    not_quote,   // первый символ литерала, не '
    escape       // после \
  } state = not_quote;
  while (from != to) {
    char const ch = *from++;
    switch (state) {
    case not_quote:
      if (ch == '\'')
        return nullptr;
      state = ch == '\\' ? escape : close_quote;
      break;
    case close_quote:
      if (ch == '\'')
        return from;
      if (ch == '\\')
        state = escape;
      break;
    case escape: // просто пропустить символ
      state = close_quote;
```

```
      break;

    default:
      assert(!"impossible FSM state occured");
    }
  }
  return nullptr; // слишком рано кончилась строка
}
```

Впрочем, аналогично состоянию open_quote можно убрать и состояние not_quote, выполнив проверку второго символа последовательности на неравенство ' перед циклом.

Пример 16.5. Символьный литерал v.3

```
char const * char_lit(
    char const * from,
    char const * to) {
  // Вытащим open_quote: данное состояние
  //   имеет смысл только на первом символе.
  if (from == to || *from++ != '\'')
    return nullptr;
  // Вытащим not_quote: данное состояние
  //   имеет смысл только на втором символе.
  if (from == to || *from == '\'')
    return nullptr;
  // Возможные состояния автомата:
  enum {
    close_quote, // ожидается закрывающая '
    escape       // после \
  } state = *from++ == '\\'? escape: close_quote;
  while (from != to) {
    char const ch = *from++;
    switch (state) {
    case close_quote:
      if (ch == '\'')
        return from;
      if (ch == '\\')
        state = escape;
      break;
    case escape: // просто пропустить символ
```

```
      state = close_quote;
      break;
    default:
      assert(!"impossible FSM state occured");
    }
  }
  return nullptr; // слишком рано кончилась строка
}
```

Теперь внутри цикла осталось всего два возможных состояния: close_quote и escape. Заменим переменную state на булевскую переменную await_quote (ее значение соответствует условию state == close_quote). Заодно совместим проверки перед циклом в один **if**.

Пример 16.6. Символьный литерал v.4

```
char const * char_lit(
    char const * from,
    char const * to) {
  // Не менее трех символов.
  // Первый символ должен быть кавычкой,
  // второй символ не должен быть кавычкой.
  if (to - from < 3
   || from[0] != '\''
   || from[1] == '\'')
    return nullptr;
  bool await_quote = from[1] != '\\';
  for (from += 2; from != to;) {
    char const ch = *from++;
    if (await_quote) {
      if (ch == '\'')
        return from;
      if (ch == '\\')
        await_quote = false;
    }
    else
      await_quote = true;
  }
  return nullptr; // слишком рано кончилась строка
}
```

Впрочем, из-за того, что случаю await_quote == **false** соответствует просто пропуск очередного символа с переходом в состояние await_quote == **true**, мы можем вообще избавиться от явного состояния автомата: в случае await_quote == **false** следует просто пропустить еще один символ.

Пример 16.7. Символьный литерал v.5

```
char const * char_lit(
    char const * from,
    char const * to) {
  // Не менее трех символов.
  // Первый символ должен быть кавычкой,
  //   второй символ не должен быть кавычкой.
  if (to - from < 3
   || from[0] != '\''
   || from[1] == '\'')
    return nullptr;
  for (++from; from != to;) {
    switch (*from++) {
    case '\\':
      if (from++ == to)
        return nullptr;
      break;
    case '\'':
      return from;
    default:
      continue;
    }
  }
  return nullptr;
}
```

Легко видеть, что теперь без **switch-case** цикл можно переписать компактнее:

Пример 16.8. Символьный литерал v.6

```
char const * char_lit(
    char const * from,
    char const * to) {
  if (to - from < 3
```

```
    || from[0] != '\''
    || from[1] == '\'')
     return nullptr;
   for (++from;;) {
     if (from == to)
       return nullptr;
     char const ch = *from++;
     if (ch == '\\' && from++ == to)
       return nullptr;
     if (ch == '\'')
       return from;
   }
}
```

Сокращение числа состояний и избавление от явного состояния в виде переменной может считаться оптимизацией (такой программный код, скорее всего, будет выполняться быстрее и будет компактнее и после компиляции). Однако вносить в него изменения в случае необходимости может стать сложнее.

Дальнейшая оптимизация быстродействия подобного кода может производиться путем обработки нескольких символов (байт) за одну итерацию.

Данный пример показывает, что от схемы автомата мы можем с помощью постепенных изменений прийти к достаточно компактному и эффективному коду. Таким образом, модель конечного автомата предоставляет нам инструмент, позволяющий в ряде случаев создавать эффективные линейные алгоритмы для обработки последовательностей данных, которые может быть сложно придумать сразу. Что еще важнее, анализировать корректность автомата проще, чем некоторого «вроде бы правильного» кода. Поэтому получить корректный и близкий к оптимальному код из автомата также может быть проще.

16.4. Классификация последовательности

Требуется классифицировать последовательность как (-1) монотонно (нестрого) убывающую, $(+1)$ монотонно (нестрого)

возрастающую, (0) состоящую из одинаковых элементов или как (2) содержащую как возрастающие, так и убывающие подпоследовательности.

Указанные числа будем использовать в качестве состояний автомата. Все состояния будем считать принимающими. На вход автомата будем подавать числа: (-1), если в очередной паре соседних элементов второй меньше первого, (0), если они равны, и $(+1)$, если второй больше первого. Данные числа могут быть результатом функции следующего вида:

```
int compare(float a, float b) {
  return a < b? +1:
         b < a? -1: 0;
}
```

Диаграмма нашего автомата-классификатора имеет вид, показанный на рис. 16.3.

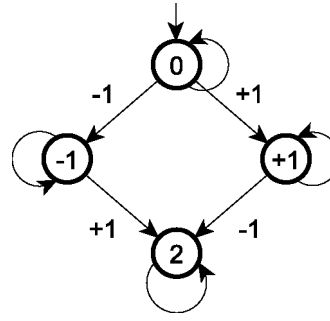


Рис. 16.3. Автомат-классификатор

Каждое из его состояний соответствует тому или иному ответу. Перепишем данную диаграмму на C++.

Пример 16.9. Классификатор последовательности

```
enum Sequence_type {
  Monotone_decreasing = -1,
```

```
  Empty_or_constant    =  0,
  Monotone_increasing  = +1,
  Not_monotone         =  2 };

Sequence_type sequence_type( // полуинтервал
     float const * from,     // [from, to)
     float const * to) {
  Sequence_type state = Empty_or_constant;
  if (from != to) {
    for (auto a = from, b = a; ++b != to; a = b) {
      int const relation = compare(*a, *b);
      switch (state) {
      case Monotone_decreasing:
        if (relation == 1)
          state = Not_monotone;
        break;
      case Empty_or_constant:
        if (relation == -1)
          state = Monotone_decreasing;
        else if (relation == 1)
          state = Monotone_increasing;
        break;
      case Monotone_increasing:
        if (relation == -1)
          state = Not_monotone;
        break;
      case Not_monotone:
        break; /* ничего не делать */
      default:
        assert(!"sequence_type: impossible state");
      }
    }
  }
  return state;
}
```

У Попробуйте преобразовать (сократить) данный код аналогично тому, как это было сделано в примере с автоматом — распознавателем символьных констант в предыдущем разделе.

16.5. Нормализация переводов строк

В качестве примера автомата-преобразователя рассмотрим конечный автомат, решающий задачу нормализации переводов строк.

Среди управляющих символов кодировки ASCII есть два символа, традиционно используемые для оформления переводов строк: код 10 *line feed* (LF, сдвинуть каретку на строку вниз, \n в C) и код 13 *carriage return* (CR, вернуть каретку в начало строки, \r в C).

Есть четыре варианта их использования: LF (Unix), CR LF (Windows и текстовые сетевые протоколы, например, HTTP), а также CR и LF CR, не используемые на современных системах. Все эти четыре варианта будем «переводить» в \n (LF), оставляя прочие символы нетронутыми.

Диаграмма автомата приведена на рис. 16.4.

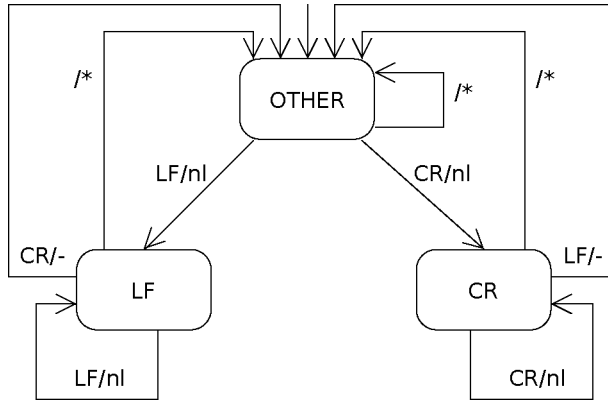


Рис. 16.4. Автомат CRLF

Обозначения на стрелках имеют вид *символ на входе* (опционально) / *символ на выходе*, где *символ на выходе* может быть «nl» (перевод строки, '\n'), «*» (вывести прочитанный символ) или «–» (ничего не выводить).

Итак, наш конечный автомат будет заменять во входном тексте все четыре варианта переводов на '\n', а прочие символы оставлять неизменными. Переведем эту схему на C++:

Пример 16.10. CRLF

```
// Возвращает указатель на символ,
// следующий за последним записанным символом.
char * crlf_normalize(
    char const * from,
    char const * end,
    char * to) {
  enum { Other, LF, CR } state = Other;
  while (from != end) {
    char const input = *from++;
    switch (state) {
    case Other:
      switch (input) {
      case '\n':
        *to++ = '\n';
        state = LF;
        break;
      case '\r':
        *to++ = '\n';
        state = CR;
        break;
      default:
        *to++ = input;
      }
      break;

    case LF:
      switch (input) {
      case '\n':
        *to++ = '\n';
        break;
```

```
        case '\r':
          state = Other;
          break;
        default:
          *to++ = input;
          state = Other;
        }
        break;

      case CR:
        switch (input) {
        case '\n':
          state = Other;
          break;
        case '\r':
          *to++ = '\n';
          break;
        default:
          *to++ = input;
          state = Other;
        }
        break;

      default:
        assert(!"crlf_normalize: impossible state");
      }
    }
    return to;
}
```

При реализации конечных автоматов вместо явно выписанного цикла и **switch** по состоянию нередко применяется простой переход по меткам (отвечающим состояниям) с помощью инструкции **goto**.

Такой код также непосредственно соответствует схеме автомата и может порождаться автоматически (программой по схеме автомата).

Его преимущество перед использованием **switch** состоит в потенциально большей производительности полученного ма-

шинного кода (роль переменной, хранящей текущее состояние автомата, играет специальный регистр процессора — указатель инструкции), что может быть заметно как раз при прямой обработке массивов символов в памяти. Более того, код получается несколько короче и даже, возможно, понятнее. Его недостаток — меньшая гибкость.

Пример 16.11. CRLF, goto

```
char * crlf_normalize(
    char const * from, char const * end,
    char * to) {
  char input;

Other:
  if (from == end) return to;
  input = *from++;
  if (input == '\n') goto LF;
  if (input == '\r') goto CR;
  *to++ = input;
  goto Other;

LF:
  *to++ = '\n';
  if (from == end) return to;
  input = *from++;
  if (input == '\r') goto Other;
  if (input == '\n') goto LF;
  *to++ = input;
  goto Other;

CR:
  *to++ = '\n';
  if (from == end) return to;
  input = *from++;
  if (input == '\n') goto Other;
  if (input == '\r') goto CR;
  *to++ = input;
  goto Other;
}
```

Конечно, данный код можно немного модифицировать, чтобы убрать **goto**, организующие явные циклы. Можно вообще убрать все **goto**, не вводя дополнительных переменных, но код от этого не станет проще, короче или понятнее.

Теперь представим, что автомат обрабатывает блок данных заранее неизвестной длины, получая их фрагментами ограниченного размера (имеет буфер ввода и буфер вывода). Таким образом, иногда необходимо приостанавливать автомат, чтобы загрузить или передать следующую порцию данных. Если взять вариант на основе **switch**, то благодаря явно хранящемуся состоянию несложно модифицировать код так, чтобы можно было продолжить обработку следующей порции данных с того состояния, в котором автомат закончил обработку предыдущей порции. Для хранения переменной состояния можно организовать объект-обертку, как в разделе 16.2. К сожалению, то же самое нельзя сказать о варианте на основе **goto**: в нем состояние не хранится в переменной.

Следующий пример представляет собой синтез версий на основе **switch** и **goto**, что позволяет обрабатывать данные в потоке с загрузкой в промежуточные буферы.

Инструкция **switch** обеспечивает «восстановление» автомата с места останова. Метки **case** могли бы быть расположены сразу после инструкций **return**, отражая логику работы (продолжение после возврата), но в примере они «подняты», чтобы входной диапазон обязательно проверялся на пустоту:

Пример 16.12. CRLF, объект

```
// Автомат, запоминающий состояние между вызовами.
class Crlf_normalizer {
  enum { Other, LF, CR } state = Other;
public:
  // Оператор () позволяет вызывать
  // объект данного класса как функцию.
  char * operator()(
      char const * from, char const * end,
      char * to) {
    char input;
```

```
    switch (state) {
    Other_:
    case Other:
      if (from == end) // выход
        return state = Other, to;
      input = *from++;
      if (input == '\n') goto LF_;
      if (input == '\r') goto CR_;
      *to++ = input;
      goto Other_;

    LF_:
      *to++ = '\n';
    case LF:
      if (from == end) // выход
        return state = LF, to;
      input = *from++;
      if (input == '\r') goto Other_;
      if (input == '\n') goto LF_;
      *to++ = input;
      goto Other_;

    CR_:
      *to++ = '\n';
    case CR:
      if (from == end) // выход
        return state = CR, to;
      input = *from++;
      if (input == '\n') goto Other_;
      if (input == '\r') goto CR_;
      *to++ = input;
      goto Other_;

    default:
      assert(!"Crlf_normalizer: impossible state");
      return to;
    }
  }
};
```

Объект класса Crlf_normalizer можно использовать так же, как функцию crlf_normalize из предыдущих вариантов. Однако, в отличие от нее, его можно применять повторно, передавая данные по частям, «разрывая» поток данных в произвольной позиции.

Задания для самопроверки

У Ответьте на следующие вопросы.

- Почему «конечный автомат» называется «конечным»?
- Является ли физический компьютер конечным автоматом?
- Может ли конечный автомат застопориться после перехода в принимающее состояние?
- В чем отличие автомата Мура от автомата Мили?

У Реализуйте конечный автомат, который принимает строку вида αabb, где α — произвольная последовательность символов, не содержащая подстроки abb.

У Реализуйте конечный автомат-преобразователь, конвертирующий двоичную запись числа в шестнадцатеричную запись того же числа.

Глава 17

Элементарные структуры данных

17.1. Статический массив

Для простоты рассматривается статический массив символов. Структуры данных, аналогичные приведенным в данном разделе, можно реализовать на основе динамического массива и элементов других типов.

17.1.1. Статический массив в качестве стека

При определении стека будем отталкиваться от стандартного определения (с. 358). Стек предоставляет четыре базовых действия: empty, push, pop, top, и, поскольку в нашей реализации массив имеет заранее ограниченный размер, проверку на заполнение full.

Реализация такого стека весьма проста.

```
class Fixed_stack {
public:
  size_t static const MAX_SIZE = 128;
  size_t size() const { return _size; }
  bool empty() const { return _size == 0; }
  bool full() const { return _size == MAX_SIZE; }
```

```
  // Положить элемент на стек.
  void push(char ch) {
    assert(!full());
    _data[_size++] = ch;
  }

  // Убрать вершину стека.
  void pop() {
    assert(!empty());
    --_size;
  }

  // Доступ к вершине стека.
  char & top() {
    assert(!empty());
    return _data[_size - 1];
  }

  char const & top() const {
    assert(!empty());
    return _data[_size - 1];
  }

private:
  char _data[MAX_SIZE]; // данные стека
  size_t _size = 0; // текущий размер стека
};
```

Например, следующий код считывает символы с потока ввода в стек и затем выводит их в обратном порядке, извлекая из стека (стек обращает порядок, см. рис. 17.1):

```
  Fixed_stack s;
  for (char c; !s.full() && cin.get(c);)
    s.push(c);
  cout << '\n';
  for (; !s.empty(); s.pop())
    cout << s.top();
```

Вместо индекса можно использовать диапазон указателей и указатель на текущую вершину стека (именно так, в частности, обычно реализован стек вызовов, указатель на вершину стека при этом хранится в регистре процессора, а диапазон проверяется посредством механизма защиты памяти).

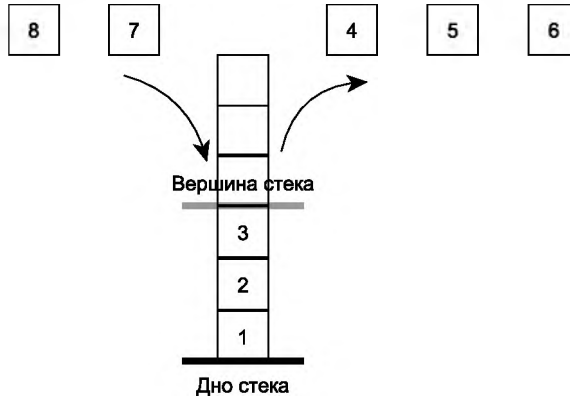


Рис. 17.1. Стек

17.1.2. Кольцевой буфер

Кольцевой буфер [*ring buffer*] — популярная структура данных, используемая для организации очередей и «окон» (при обработке последовательностей данных). Ее преимущество заключается в простоте и быстродействии. Недостаток — фиксированный ограниченный размер.

```
class Ring_buffer {
public:
  size_t static const SIZE = 4;
  char data[SIZE] {};

  // Положить элемент на следующую позицию в буфере.
  void put(char item) {
    data[_pos++] = item;
```

```
    if (_pos == SIZE)
      _pos = 0;
    // или _pos %= SIZE
  }

  char & operator[](size_t index) {
    return data[(_pos + index) % SIZE];
  }

  char const & operator[](size_t index) const {
    return data[(_pos + index) % SIZE];
  }

private:
  size_t _pos = 0;
};
```

Функция put добавляет элемент, затирая предыдущие элементы «по кругу». Массив data сделан открытым, чтобы было удобно обрабатывать содержимое целиком, если не требуется доступ в порядке добавления элементов (для чего предоставлен оператор []).

Например, можно сделать бегущее среднее (групп по 4 байта в нашем случае):

```
  Ring_buffer window;
  for (size_t f = Ring_buffer::SIZE; f-- != 0;)
    window.put(cin.get()); // заполнить
  do {
    int sum = 0;
    for (int item: window.data)
      sum += item;
    cout.put(char(sum/Ring_buffer::SIZE));
    if (char c; cin.get(c))
      window.put(c);
  } while (cin);
```

У Можно ли в данном примере избавиться от суммирования всех элементов на каждом шаге?

На основе кольцевого буфера несложно реализовать структуру данных, называемую *очередью*.

Очередь [*queue*] — линейная структура данных, позволяющая добавлять и удалять элементы по одному, при этом элементы удаляются в том же порядке, в котором добавлялись («первым зашел — первым вышел» [*FIFO*], см. рис. 17.2), и возможен доступ к первому добавленному элементу (*начало очереди* [*queue front*]).

Очередь предлагает набор операций, аналогичный стеку: push (добавить элемент в конец очереди), pop (убрать элемент из начала очереди), front (доступ к элементу в начале очереди), empty (проверить очередь на пустоту).

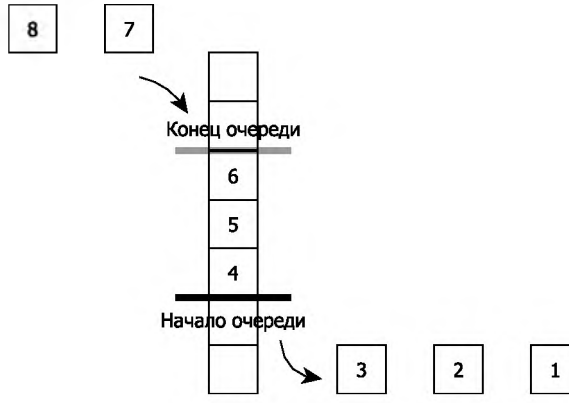


Рис. 17.2. Очередь

Из кольцевого буфера можно сделать очередь, добавив поле для хранения текущего размера очереди _sz:

```
class Ring_queue {
public:
  size_t static const SIZE = 32;
  size_t size() const { return _sz; }
  bool empty() const { return _sz == 0; }
  bool full() const { return _sz == SIZE; }
```

```
  void push(char item) {
    assert(!full());
    data[_wp++] = item;
    if (_wp == SIZE)
      _wp = 0;
    ++_sz;
  }

  void pop() {
    assert(!empty());
    --_sz;
  }

  char & front() {
    assert(!empty());
    return data[(_wp - _sz) % SIZE];
  }

  char const & front() const {
    assert(!empty());
    return data[(_wp - _sz) % SIZE];
  }

private:
  size_t _wp = 0, _sz = 0;
  char data[SIZE] {};
};
```

Двунаправленная очередь [*double ended queue, deque*] — линейная структура данных, позволяющая добавлять и удалять элементы с двух концов. Таким образом, двунаправленная очередь может работать и как очередь, и как стек, причем в двух направлениях.

Для реализации двунаправленной очереди можно взять нашу кольцевую очередь и добавить операции:

```
struct Ring_deque {
public:
  size_t static const SIZE = 32;
  bool empty() const { return _sz == 0; }
```

```
bool full() const { return _sz == SIZE; }
size_t size() const { return _sz; }

// Добавить элемент в начало очереди.
void push_front(char item) {
  assert(!full());
  ++_sz;
  data[_front()] = item;
}

// Добавить элемент в конец очереди.
void push_back(char item) {
  assert(!full());
  data[_wp++] = item;
  if (_wp == SIZE)
    _wp = 0;
  ++_sz;
}

// Убрать элемент с начала очереди.
void pop_front() {
  assert(!empty());
  --_sz;
}

// Убрать элемент с конца очереди.
void pop_back() {
  assert(!empty());
  _wp = _back();
  --_sz;
}

// Доступ к элементу в начале очереди.
char & front() {
  assert(!empty());
  return data[_front()];
}

char const & front() const {
  assert(!empty());
```

```
    return data[_front()];
  }

  // Доступ к элементу в конце очереди.
  char & back() {
    assert(!empty());
    return data[_back()];
  }

  char const & back() const {
    assert(!empty());
    return data[_back()];
  }

private:
  size_t _wp = 0, _sz = 0;
  char data[SIZE] {};

  size_t _front() const {
    return (_wp - _sz) % SIZE;
  }

  size_t _back() const {
    return (_wp == 0? SIZE: _wp) - 1;
  }
};
```

В качестве забавного примера использования двунаправленной очереди можно привести код, раскидывающий символы ввода влево-вправо в зависимости от четности позиции:

```
int main() {
  Ring_deque rd;
  for (char ch;;) {
    if (rd.full() || !cin.get(ch))
      break;
    rd.push_back(ch);
    if (rd.full() || !cin.get(ch))
      break;
    rd.push_front(ch);
  }
```

```
  while (!rd.empty()) {
    cout.put(rd.front());
    rd.pop_front();
  }
}
```

17.2. Связанный список

Связанный список [*linked list*] — структура данных, состоящая из *звеньев* [*links*], каждое из которых указывает либо на одно другое звено (*односвязный список*, рис. 17.3), либо на два звена (предыдущее и следующее — *двусвязный список*, рис. 17.4).

Линейный список — связанный список, который или пуст, или обязательно содержит звено, у которого нет предыдущего, и звено, у которого нет следующего.

Признаком отсутствия предыдущего или следующего звена обычно служит равенство соответствующего указателя нулевому указателю.

17.2.1. Линейный список в качестве стека

На основе односвязного списка несложно реализовать стек потенциально неограниченного размера.

Звено обычно реализуется как структура вида:

```
struct Link {
  Item value;
  Line * next = nullptr;
};
```

Объект списка хранит указатель на «голову» — последнее созданное звено. Этот указатель нулевой, если список пуст.

Добавление и удаление звена «в голове» списка:

```
struct List {
  Link * head = nullptr;
```

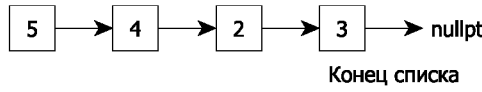


Рис. 17.3. Линейный односвязный список

```
  bool empty() const { return head == nullptr; }

  void push(Item item) {
    head = new Link { item, head };
  }

  void pop() {
    assert(!empty());
    auto new_head = head->next;
    delete head;
    head = new_head;
  }

  ~List() {
    while (!empty())
      pop();
  }
};
```

Данный список пока не совсем хорош, поскольку он нарушает правило трех. Можно реализовать аналогичную конструкцию на основе std::unique_ptr (но деструктор все равно желательно оставить свой, поскольку иначе можно получить переполнение стека вызовов при удалении большого списка):

```
#include <cstddef>
#include <cassert>
#include <memory>
using namespace std;

using Item = double;
class L1_stack {
public:
```

```
  bool empty() const { return _head == nullptr; }

  void push(Item item) {
    _head = make_unique<Link>(item, move(_head));
  }

  void pop() {
    assert(!empty());
    _head.reset(_head->next.release());
  }

  Item & top() {
    assert(!empty());
    return _head->value;
  }

  Item const & top() const {
    assert(!empty());
    return _head->value;
  }

  void clear() {
    while (!empty())
      pop();
  }

  ~L1_stack() { clear(); }

private:
  struct Link {
    Item value;
    unique_ptr<Link> next;

    Link(Item value, unique_ptr<Link> next)
      : value(value), next(move(next)) {}
  };

  unique_ptr<Link> _head;
};
```

17.2.2. Двусвязный список

Звено двусвязного списка может иметь вид:

```
struct Link {
  Item value;
  Link * prev, * next;
};
```

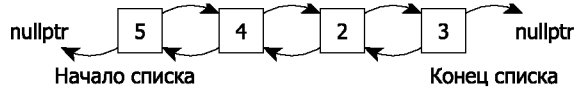


Рис. 17.4. Линейный двусвязный список

Соответственно, вставка и удаление элемента требуют корректной привязки как указателей prev, так и указателей next. Можно считать, что двусвязный список есть два параллельных односвязных списка, проходящих по одним и тем же звеньям в противоположных направлениях.

В целом, работа с двусвязным списком проще, чем с односвязным: имея только указатель на звено, можно удалить это звено, вставить звено перед ним или после него.

В случае линейного списка поля prev первого звена и next последнего звена хранят нулевые указатели.

У Реализуйте двунаправленную очередь на основе двусвязного списка.

17.2.3. Кольцевой односвязный список

Кольцевой список — связанный список, в котором последнее звено указывает на первое звено (и наоборот, в случае двусвязного списка), см. рис. 17.5.

Интерес представляет тот факт, что кольцевой односвязный список можно использовать и как стек, и как очередь. Для этого можно считать, что «голова» — это последний элемент (конец очереди). Поскольку список — кольцевой, послед-

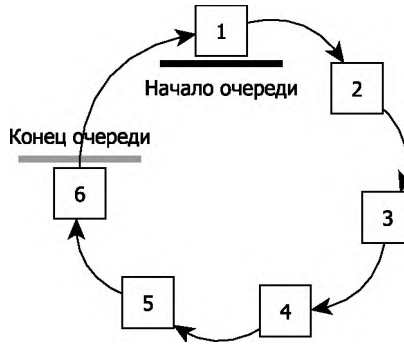


Рис. 17.5. Кольцевой односвязный список

ний элемент указывает на первый, и мы легко получаем доступ к началу очереди (по совместительству, вершине стека). Чтобы добавить элемент в начало, достаточно вставить его после головы. Чтобы добавить элемент в конец, достаточно вставить его в начало и сдвинуть голову вперед.

Полноценная двусторонняя очередь у нас не получится, поскольку невозможно, не проходя весь список, удалить элемент в конце очереди. Итак, код:

```
class L1_ring {
public:
  bool empty() const { return _head == nullptr; }

  void push_front(Item item) {
    if (empty())
      return _make_singleton(item);
    _insert_after(_head, new Link{item});
  }

  void push_back(Item item) {
    push_front(item);
    _head = _head->next;
  }
```

```
void pop_front() {
  assert(!empty());
  if (!_pop_singleton())
    _delete_after(_head);
}

Item & back() {
  assert(!empty());
  return _head->value;
}

Item const & back() const {
  assert(!empty());
  return _head->value;
}

Item & front() {
  assert(!empty());
  return _head->next->value;
}

Item const & front() const {
  assert(!empty());
  return _head->next->value;
}

void clear() {
  while (!empty())
    pop_front();
}

void swap(L1_ring & other) {
  std::swap(_head, other._head);
}

// Правило пяти.
~L1_ring() { clear(); }
L1_ring() = default;
L1_ring(L1_ring const&) = delete;
L1_ring & operator=(L1_ring const&) = delete;
```

```
  L1_ring(L1_ring && other)
    : _head(other._head) { other._head = nullptr; }

  L1_ring & operator=(L1_ring && other) {
    L1_ring(move(other)).swap(*this);
    return *this;
  }

private:
  struct Link {
    Item value;
    Link * next;
  } * _head = nullptr;

  void _make_singleton(Item value) {
    assert(empty());
    _head = new Link { value };
    _head->next = _head;
  }

  bool _pop_singleton() {
    if (_head != _head->next)
      return false;
    delete _head;
    _head = nullptr;
    return true;
  }

  void _insert_after(Link * link, Link * new_link) {
    new_link->next = link->next;
    link->next = new_link;
  }

  void _delete_after(Link * link) {
    auto del_link = link->next;
    assert(del_link != link);
    link->next = del_link->next;
    delete del_link;
  }
};
```

17.3. Двоичное дерево

Дерево [*tree*] — структура данных, состоящая из *узлов* [*nodes*], каждый из которых может ссылаться на другие узлы, при этом узлы можно пометить натуральными числами таким образом, что:

- только один узел (*корень* [*root*]) имеет метку 1;
- метка всякого узла строго меньше меток узлов, на которые он ссылается;
- на каждый узел кроме корня ссылается ровно один другой узел.

Узлы, на которые ссылается узел, называются его *потомками* [*children*], а сам он — их *родителем* [*parent*]. Узел, не имеющий потомков, называется *листом* [*leaf*].

Каждый раз, переходя от родителя к потомку, мы можем назначать потомку метку, на единицу большую метки родителя. Такой способ помечивания даст наименьшее возможное значение максимальной метки, которое называется *глубиной* [*depth*]. Другим словами, глубина есть максимальное из расстояний от корня дерева до его листьев.

Должно быть очевидно, что дерево есть *рекурсивная структура данных*: дерево это или лист, или набор деревьев («поддеревьев»). Линейный односвязный список есть частный случай дерева, в котором каждый узел может содержать не более одного поддерева.

Двоичное дерево [*binary tree*] — дерево, у каждого узла которого может быть не более двух (непосредственных) потомков.

Двоичное дерево поиска [*binary search tree, BST*] — двоичное дерево, каждый узел которого содержит некоторое значение. При этом значения всех узлов одного поддерева («левого») должны быть меньше значения родителя, а значения всех узлов другого поддерева («правого») — больше (или равны, если допускаются повторы значений) значения родителя, см. рис. 17.6.

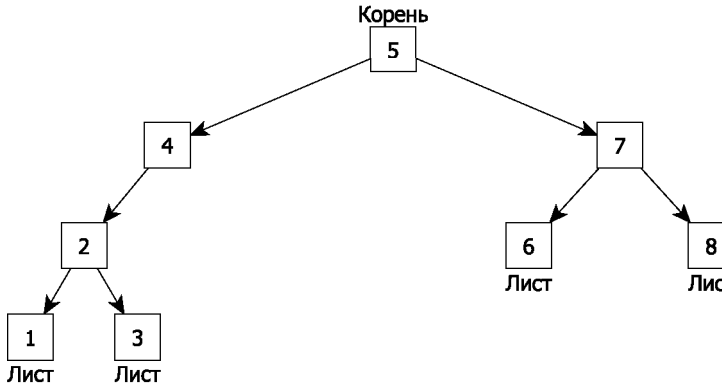


Рис. 17.6. Двоичное дерево поиска

Двоичное дерево поиска позволяет выполнять вставку, удаление и поиск значений за число сравнений, ограниченное глубиной дерева, а не общим числом его элементов. Глубина двоичного дерева ограничена снизу величиной $\lceil \log_2 N \rceil + 1$, где N есть число узлов.

Операции над деревом удобно определять рекурсивно:

```
// Узел двоичного дерева поиска.
struct Bst_node {
  Item value;
  unique_ptr<Bst_node> left, right;

  Bst_node() = default;
  // Конструктор, принимающий значение узла.
  explicit Bst_node(Item value)
    : value(value) {}
};

// Вычисление глубины дерева.
size_t depth(Bst_node * root) {
  return root?
    max(depth(root->left.get()),
```

```
            depth(root->right.get())) + 1
    : 0;
}

// Поиск элемента в дереве.
Bst_node * find(Bst_node * root, Item item) {
  if (root) {
    if (item < root->value)
      return find(root->left.get(), item);
    if (root->value < item)
      return find(root->right.get(), item);
  }
  return root;
}

// Вставка элемента в дерево.
Bst_node * insert(Bst_node * root, Item item) {
  if (root) {
    unique_ptr<Bst_node> * where = nullptr;
    if (item < root->value)
      where = &root->left;
    else if (root->value < item)
      where = &root->right;
    else
      return root;
    if (*where)
      return insert(where->get(), item);
    *where = make_unique<Bst_node>(item);
    return where->get();
  }
  return root;
}
```

У Напишите рекурсивную функцию, выводящую содержимое дерева в порядке сортировки (сначала вывести левое поддерево, затем — значение узла (корня), затем — правое поддерево).

В случае функций find и insert имеем хвостовые вызовы, поэтому их легко переписать, используя циклы.

Другое дело — функция depth, функция вывода дерева из упражнения выше и функция удаления дерева (деструктор по умолчанию Bst_node рекурсивен).

В случае дерева все равно можно избавиться от рекурсии, не используя дополнительные структуры данных (скажем, отдельный стек для хранения отложенных узлов). Например, деструктор можно написать так, чтобы он собирал узлы в очередь, организуя на них односвязный список по указателям на потомков:

```
~Bst_node() {
// Переменная del хранит указатель на удаляемый узел,
// head хранит указатель на начальное звено очереди,
// grow хранит указатель на указатель,
// куда в очередь должен быть поставлен следующий узел.
  for (unique_ptr<Bst_node>
         del, head, *grow = &head;;) {
    switch ( // Вычислим вариант от 0 до 3.
        ((left != nullptr) << 1)
      + (right != nullptr)) {
    case 0: // нет потомков, берем со стека
      if (!head) return;
      del = move(head->left);
      right = move(head->right);
      head = move(right);
      break;
    case 1: // есть только правый потомок
      del = move(right);
      break;
    case 2: // есть только левый потомок
      del = move(left);
      break;
    case 3: // есть оба потомка
      del = move(left);
      *grow = move(right);
      // Новый конец очереди — самый правый лист.
      while (*grow)
        grow = &((*grow)->right);
      break;
```

```
      default:
        assert(!"~Bst_node: impossible");
      }

      // Забрать потомков у удаляемого узла.
      left = move(del->left);
      right = move(del->right);
    }
}
```

Конечно, деструктору не нужно сохранять структуру дерева, так как оно все равно будет уничтожено, а что делать при реализации операций обхода дерева без изменения его структуры? На этот вопрос есть, по крайней мере, три возможных ответа:

- реализовать *сбалансированное* дерево (глубина которого всегда пропорциональна логарифму количества узлов) и пользоваться рекурсией (глубина которой ограничена сверху глубиной дерева);

- отказаться от неизменности структуры, изменяя ее в процессе обхода, но делать это таким образом, чтобы ее можно было восстановить. Этого можно добиться с помощью *вращений*[138];

- добавить в узел указатель на родителя и построить процедуру обхода дерева в форме конечного автомата.

Приведем пример реализации обхода дерева с помощью возвращения по указателям на родителей[139]. Итак, определим класс Bst — «двоичное дерево поиска»:

---

[138] См.: *Степанов А. А., Мак-Джонс П.* Начала программирования. М., 2011. Гл. 8. Координаты с изменяемыми последователями.

[139] Алгоритм позаимствован из книги Степанова и Мак-Джонса (см.: *Степанов А. А., Мак-Джонс П.* Начала программирования. С. 130–133).

```
class Bst {
public:
  bool empty() const { return _root == nullptr; }
  size_t size() const;
  size_t depth() const;

  ~Bst() {
    clear();
    assert(empty());
  }

  bool contains(Item const & item) const;
  bool insert(Item const & item);
  void clear();
  void inorder(void (&visit)(Item const&)) const;

private:
  struct Node {
    Item value;
    Node *left   = nullptr,
         *right  = nullptr,
         *parent = nullptr;

    Node() = default;
    Node(Item const & value,
         Node * parent = nullptr)
      : value(value), parent(parent) {}
  } * _root = nullptr;
  Node const * _front() const;
  // Стадия посещения (до узла, в узле, после узла).
  enum Step { Pre, In, Post };
  // Шаг обхода, возвращает изменение по высоте.
  int _step(Step & step, Node *& cur) const;
};
```

Удаление дерева теперь реализовать проще (теперь изменение структуры заключается лишь в замене ссылки на удаляемый левый потомок его правым потомком):

```
Bst::Node const * Bst::_front() const {
  assert(_root != nullptr);
```

```
  for (auto cur = _root;;) {
    auto next = cur->left;
    if (next == nullptr)
      return cur;
    cur = next;
  }
}

void Bst::clear() {
  while (_root) {
    for (auto cur = _front(); cur != _root;) {
      auto par = cur->parent;
      cur->parent->left = cur->right;
      delete cur;
      cur = par;
    }

    auto new_root = _root->right;
    delete _root;
    _root = new_root;
  }
}
```

Шаг обхода дерева представляет собой шаг работы конечного автомата с состоянием step (Pre — состояние «узел посещен впервые», In — «левое поддерево обработано», Post — «правое поддерево обработано»):

```
int Bst::_step(Bst::Step &step, Node *&cur) const {
  assert(cur && (step != Post || cur->parent));
  switch (step) {
  case Pre:
    if (auto left = cur->left) {
      cur = left; // спуск
      return 1;   // влево
    } else {
      step = In; // нет левого
      return 0;  // поддерева
    }
  case In:
    if (auto right = cur->right) {
```

```
      step = Pre;  // спуск
      cur = right; // вправо
      return 1;
    } else {
      step = Post; // нет правого
      return 0;    // поддерева
    }
  case Post:
    if (cur->parent->left == cur)
      step = In; // если спускались влево
    cur = cur->parent; // подъем к
    return -1;         // родителю
  }
}

void Bst::inorder(void(&visit)(Item const&)) const {
  if (empty()) return;
  Step step = Pre;
  auto cur = _root;
  do {
    _step(step, cur);
    if (step == In)
      visit(cur->value);
  } while (cur != _root || step != Post);
}
```

Теперь можно определить функцию, выводящую содержимое дерева в порядке сортировки, например, так:

```
void print_tree(Bst const & tree) {
  struct Print {
    static void item(Item const & item) {
      cout << item << ' '; }};
  tree.inorder(Print::item);
  cout << '\n';
}
```

На основе той же конструкции легко реализовать вычисление размера и глубины дерева:

```
size_t Bst::size() const {
  if (empty()) return 0;
```

```
  size_t sz = 1;
  Step step = Pre;
  auto cur = _root;
  do {
    _step(step, cur);
    sz += step == Pre;
  } while (cur != _root || step != Post);
  return sz;
}

size_t Bst::depth() const {
  if (empty()) return 0;
  size_t cur_depth = 2, max_depth = 0;
  Step step = Pre;
  auto cur = _root;
  do {
    cur_depth += _step(step, cur);
    max_depth = max(max_depth, cur_depth);
  } while (cur != _root || step != Post);
  return max_depth - 1;
}
```

17.4. Упакованное двоичное дерево

Упакованное дерево — дерево, значения узлов которого хранятся в массиве, а связи между узлами заключаются в соотношениях между индексами (указатели не хранятся).

В случае двоичного дерева обычно используется следующее соотношение: если i — индекс родителя, то индексы левого и правого потомков, соответственно, равны $2i+1$ и $2i+2$. Корень имеет индекс 0.

Упакованное двоичное дерево обычно используется для реализации *двоичной пирамиды*, в частности, в алгоритме пирамидальной сортировки (с. 457) и при организации очереди с приоритетом (достаем элементы не в порядке их добавления, а в порядке сортировки).

Задания для самопроверки

У Используя кольцевой буфер, реализуйте бегущее среднее 8 последних введенных чисел типа **float**.

У Реализуйте стек поверх динамического массива.

У Реализуйте двунаправленную очередь[140] (можно добавлять и удалять элементы и с начала, и с конца) поверх динамического массива.

У Реализуйте добавление и удаление элемента односвязного списка в произвольной позиции (по указателю на предыдущее звено).

У Пусть есть односвязный список, который может быть трех видов: линейный, кольцевой и ρ-образный.

Под ρ-образным списком подразумевается «кольцо с ручкой»: в начале находится линейная последовательность, в конце которой стоит звено, указывающее на какое-то из звеньев этой последовательности. Можно считать ρ-образный список обобщением линейного (кольцевая часть размера нуль) и кольцевого («ручка» размера нуль) списков.

Задача: используя постоянное количество памяти за линейное по числу звеньев списка время, определить, к какому из трех видов относится список.

Как определить размеры ручки и кольца?

[140] Двунаправленную очередь кратко называют *дек* — англ. *deque*, сокращение от *double ended queue*.

Глава 18

Дополнительный материал

18.1. Сравнение C++ с Pascal

Данный раздел предназначен для облегчения начала программирования на C++ тем, кто знаком с языком программирования Pascal. Далее перечисляются примеры кода на Pascal и дается возможный аналог на C++ или комментарий.

```
program MyProg;
unit MyUnit;
library MyLib;
interface
implementation
```

Данный код не имеет прямого аналога на C++. C++ не предоставляет средств для объявления модулей. Модулем считается файл с исходным кодом.

```
uses MyUnit;
```

Неточным аналогом будет:

```
#include "my_unit.h"
```

```
(* комментарий *)
{ комментарий }
```

Синтаксис многострочного комментария C++:

```
/* комментарий */
```

Составной оператор в Pascal

```
begin
end
```

соответствует блоку в C++:

```
{
}
```

Минимальная программа на Pascal:

```
program Empty; begin end.
```

Минимальная программа на C++:

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

Строковая константа в Pascal:

```
'I don''t know what''s this: "'#199'".'
```

Строковая константа в C++:

```
"I don't know what's this: \"\xC7\"."
```

Определение функции в Pascal:

```
function sqr(x: Integer): Integer;
begin
  sqr := x * x
end;
```

В C++ тип указывается перед именем переменной, параметра или функции. Вместо присваивания имени функции результат указывается как «аргумент» инструкции **return**, выполнение которой приводит к выходу из тела функции:

```
int sqr(int x) {
  return x * x;
}
```

Определение процедуры в Pascal:

```
procedure hello(const name: String);
begin
  write('Hello, ');
  writeln(name)
end;
```

В C++ нет разделения на процедуры и функции. Процедуры — это функции, не возвращающие значения. Для указания этого факта ставится тип **void**:

```
void hello(char const * name) {
  std::cout << "Hello, "
            << name << '\n';
}
```

В Pascal могут существовать процедура и функция с одним и тем же именем, поскольку вызов процедуры — «оператор» (инструкция), а вызов функции — «выражение». Таким образом, процедуры и функции существуют в «разных мирах» и не пересекаются. В C++ есть только функции и вызов функции является выражением.

Определение переменных в Pascal:

```
const
    meaning = 42;
var
    i, j: Integer;
    a, b: Single;
    x, y: Double;
    enabled: Boolean;
```

В C++ нет секций const и var. Переменные определяются в произвольном нужном месте функции, их область видимости ограничена блоком. Глобальные переменные определяются вне функций и доступны фунциям, определенным ниже по тексту. Константность является частью типа и включается в описание типа. Значения переменных по умолчанию не определены:

```
int const meaning = 42;
```

```
int i, j;
float a, b;
double x, y;
bool enabled;
```

Запись числовых и символьных констант в Pascal:

```
var
    bin: Integer = %1001;
    hex: Integer = $ABCD;
    ch: Char = chr(65);
    codA: Integer = ord('A');
```

Аналоги в C++:

```
int bin = 0b1001;
int hex = 0xABCD;
int oct = 07337;
char ch = 65;
int codA = 'A';
```

Еще пример функции на Pascal:

```
function length(x, y: Double): Double;
begin
  length := sqrt(x * x + y * y)
end;
```

В C++ функция sqrt, вычисляющая квадратный корень, объявлена в cmath. При определении функции необходимо указывать тип для каждого параметра функции:

```
#include <cmath>
double length(double x, double y) {
  return std::sqrt(x * x + y * y);
}
```

Ветвление в Pascal:

```
if a < b then
  write('a is too low!');
```

Аналог в C++ (условие обязательно помещается в скобки, then не пишется):

```
if (a < b)
  std::cout << "a is too low!";
```

Еще ветвление:

```
if x <> 0 then
  y := 0
else if y = 0 then
  x := 1;
```

То же самое в C++:

```
if (x != 0)
  y = 0;
else if (y == 0)
  x = 1;
```

Одним из существенных отличий является то, что в C++ присваивание является операцией и может быть частью выражения. В частности, это позволяет сделать цепочку присваиваний:

```
// Обнулить a, b, c, d:
a = b = c = d = 0;
```

Еще операции, которые записываются по-другому в Pascal:

```
inc(i);
dec(j);
xy := x ** y;
quot := n div p;
rem  := n mod p;
flag := (a = 1) or (a = 2) and not waiting;
```

и в C++:

```
++i;
--j;
xy = std::pow(x, y); // #include <cmath>
quot = n / p; // целочисленное для целых операндов
rem  = n % p;
flag = a == 1 || a == 2 && !waiting;
// and, or, not в C++ также допустимы
```

Поразрядные операции в Pascal:

not and or xor shl shr

записываются в C++ иначе:

~ & | ^ << >>

Функции ord, succ, pred, low, high не имеют прямых аналогов в C++.

Метки и goto в Pascal:

```
procedure DontRunMe;
label forever;
begin
  forever: write('*');
  goto forever
end;
```

В C++ секции label нет, метки для goto не требуется объявлять заранее. Вызов функции, не принимающей параметров, требует наличия скобок:

```
void dont_run_me() {
forever:
  std::cout << '*';
  goto forever;
}
```

Выбор варианта в Pascal:

```
case season of
  0: write('winter');
  1: write('spring');
  2: write('summer');
  3: write('autumn');
  else write('error');
end;
```

Аналог в C++ (важно наличие **break**):

```
switch (season) {
case 0:
  std::cout << "winter";
```

```
  break;
case 1:
  std::cout << "spring";
  break;
case 2:
  std::cout << "summer";
  break;
case 3:
  std::cout << "autumn";
  break;
default:
  std::cout << "error";
}
```

Более сложный пример — решение квадратного уравнения.
Pascal:

```
function SolveQuadratic(
   a, b, c: Double;
   var x1, x2: Double): Boolean;
var d: Double;
begin
  result := false;
  if a <> 0 then
  begin
    d := b * b - 4.0 * a * c;
    if d >= 0 then
    begin
      x1 := (-b - d) / (2.0 * a);
      x2 := (-b + d) / (2.0 * a);
      result := true;
    end
  end
end;
```

C++:

```
bool SolveQuadratic(
   double a, double b, double c,
   double & x1, double & x2) {
  if (a == 0)
    return false;
```

```
  double d = b * b - 4.0 * a * c;
  if (d < 0)
    return false;
  d = std::sqrt(d);
  x1 = (-b - d) / (2.0 * a);
  x2 = (-b + d) / (2.0 * a);
  return true;
}
```

Цикл с предусловием в Pascal:

```
while an < b do
begin
  an := an + a;
  inc(n)
end;
```

Аналогичная конструкция в C++:

```
while (an < b) {
  an += a;
  ++n;
}
```

Цикл с постусловием в Pascal:

```
repeat
  PerformOperation;
  write('Repeat? Y/N ');
  read(ch);
until (ch = 'n') or (ch = 'N');
```

Похожая конструкция в C++ (в конце — условие продолжения цикла):

```
do {
  PerformOperation();
  std::cout << "Repeat? Y/N ";
  ch = std::cin.get();
} while (ch != 'n' && ch != 'N');
```

Цикл со счетчиком в Pascal:

```
for i := 1 to 9 do
  writeln(i * i);
for i := 9 downto 1 do
  writeln(i * i * i);
```

Похожая конструкция в C++:

```
for (i = 1; i < 10; ++i)
  std::cout << i * i << '\n';
for (i = 9; i > 0; --i)
  std::cout << i * i * i << '\n';
```

Статические массивы в Pascal:

```
var
  a: array[0..10] of Integer;
  a2: array[0..10, 0..20] of Integer;
begin
 a2[5, 6] := 56;
```

Статические массивы в C++ (нумерация индексов всегда с нуля, в объявлении указывается только размер, двумерный массив задается как массив массивов):

```
int a[11];
int a2[11][21];
// какой-то код
a2[5][6] = 56;
```

Объявление синонима типа в Pascal:

```
type
  MyInt = Integer;
  MyArr = array[0..9] of Double;
```

Объявление синонима типа в C++ (нет секции type):

```
using MyInt = int;
using MyArr = double[10];
```

Тип-перечисление в Pascal:

```
type
  Season = (Winter, Spring, Summer, Autumn);
```

Тип-перечисление в C++:

```
enum Season { Winter, Spring, Summer, Autumn };
```

Прямых аналогов типа-диапазона и типа-множества (set) в C++ нет.

Запись в Pascal:

```
type
  Point = record
    x, y: Single;
  end;
```

соответствует структуре в C++:

```
struct Point {
  float x, y;
};
```

Вариантная запись в Pascal:

```
type
  RGBA = record case Boolean of
    false: data: Longword;
    true: r, g, b, a: Byte;
  end;

  PointKind = (Cartesian, Polar);
  Point = record case
    kind: PointKind of
    Cartesian: x, y: Double;
    Polar: rho, phi: Double;
  end;
```

Аналог на C++ на основе объединения и структуры:

```
union RGBA {
  uint32_t data;
  struct {
    uint8_t r, g, b, a;
  };
};

enum PointKind {Cartesian, Polar};
```

```
struct Point {
  union {
    struct { double x, y; };
    struct { double rho, phi; };
  };
  PointKind kind;
};
```

Проверка тега (kind) в случае C++ должна быть реализована программистом.

Прямого аналога файлового типа в C++ нет. Вместо него можно использовать классы стандартной библиотеки, определенные в заголовочном файле **fstream**.

Указатели в Pascal:

```
var
  n: Integer = 0;
  p: ^Integer = nil;
begin
  p := @n; { @пусть p указывает на n@ }
  p^ := 1; { @теперь n = 1@ }
```

Указатели в C++:

```
int n = 0;
int * p = nullptr;
p = &n; // пусть p указывает на n
*p = 1; // теперь n == 1
```

Указатель на массив (строку):

```
var
  a: String[100];
  p: ^String[100];
begin
  p := @a; { @пусть p указывает на a@ }
  p^[50] := 'A'; { @теперь a[50] = 'A'@ }
```

Указатель на массив в C++:

```
char a[100];
char *p = a; // p указывает на a
p[49] = 'A'; // теперь a[49] == 'A'
```

Статические массивы неявно приводятся к указателям на первый элемент. К самому указателю можно обращаться как к массиву, используя операцию []. Так как в C++ a — просто массив символов, отсчет начинается с нуля, а не с единицы, как в примере на Pascal (поэтому в примере 49, а не 50).

В C++ нет встроенного строкового типа. Для оперирования строками в стандартной библиотеке C++ имеются классы string и string_view, определенные в одноименных заголовочных файлах.

Бестиповый указатель в Pascal:

```
type
  IntPointer = ^Integer;
var
  pn: IntPointer;
  pp: Pointer;
begin
  pp := pn;
  pn := IntPointer(pp);
```

Бестиповый указатель в C++:

```
using IntPointer = int*;
IntPointer pn;
void * pp = pn;
pn = IntPointer(pp);
```

Динамическая память в Pascal:

```
var
  p: ^Integer;
begin
  new(p);
  p^ := 23;
  { ... }
  dispose(p);
```

Динамическая память в C++(**new** позволяет сразу инициализировать новосозданную переменную нужным значением, **delete** не изменяет значения указателя):

```
int * p = nullptr;
p = new int(23);
// ...
delete p;
p = nullptr; // это делать необязательно
```

Динамический массив в Pascal:

```
var
  n: Integer;
  a: ^array[0..10000] of Integer;
begin
  n := ...;
  GetMem(a, n * 4);
  { ... }
  FreeMem(a, n * 4);
```

Динамический массив в C++:

```
size_t n;
int * a;
n = ...;
a = new int[n];
// ...
delete[] a; // размер не нужен
```

На практике в качестве динамического массива обычно используют класс стандартной библиотеки C++ vector.

Процедурный тип (функциональный аналогично) в Pascal:

```
type
  StrProc = procedure(N: String);
var
  hello: StrProc;
  name: String;
  procedure HelloEn(N: String);
  begin write('Dear ', N, ',') end;
  procedure HelloRu(N: String);
  begin write('Здравствуйте, ', N, '!') end;
begin
  hello := @HelloEn;
  hello(name);
```

```
  hello := @HelloRu;
  hello(name);
```

Указатель на функцию в C++:

```
using StrProc = void (*)(const char *n);
StrProc hello;
char name[100];

void HelloEn(const char *n) {
  cout << "Dear " << n << ',';
}

void HelloRu(const char *n) {
  cout << "Здравствуйте, " << n << '!';
}

int main() {
  hello = HelloEn;
  hello(name);
  hello = HelloRu;
  hello(name);
```

Предварительное объявление типа в Pascal:

```
type
  PLink = ^TLink;
  TLink = record
    data: Pointer;
    next: PLink;
  end;
```

То же самое в C++ можно записать так:

```
struct TLink;
using PLink = TLink*;
struct TLink {
  void *data;
  PLink next;
};
```

На самом деле в данном примере в случае C++ предварительное объявление не нужно, поскольку заголовок структуры считается таковым, а значит, можно записать просто:

```
struct Link {
  void *data;
  Link *next;
};
```

Доступ к полю записи через указатель на объект этой записи:

```
var
  list: PLink;
  data: PLink;
begin
  list^.data := data;
```

Доступ к полю структуры через указатель на объект этой структуры:

```
TLink *list, *data;
// ...
list->data = data;
// То же, что
// (*list).data = data;
```

В отличие от операции «точка», в C++ можно определять операцию «стрелка» для своих типов.

18.2. Препроцессор

Директивы препроцессора занимают, по крайней мере, одну строчку и начинаются с символа #. Рассмотрим стандартные директивы препроцессора.

include — препроцессор заменяет строчку директивы include на содержимое файла, указанного в директиве.

define — вводит определение *макроподстановки*[141] (текстовой замены). Существует два варианта макроподстановки: слово и функция.

[141] Также «макрос», англ. *macro*, *macroinstruction*, от др.-греч. *μακρός* — «длинный», т. е. «длинная инструкция». Имелось в виду, что это короткая запись длинного куска кода.

Макроподстановка-слово определяет простую подстановку без параметров.

Пример 18.1. Pascal-скобки

```
#define BEGIN {
#define END }
int main() BEGIN
  // BEGIN заменяется на {
  // END заменяется на }
END
```

Замена может быть пустой (на пустую строку). Например:

```
#define WIN
```

Обычно так делается, чтобы было определено некое имя (здесь WIN), см. далее директивы if, ifdef, ifndef.

Препроцессор различает большие и маленькие буквы. Таким образом, BEGIN, Begin, begin и т. д. — разные имена. Макроподстановкам принято давать имена, записанные большими буквами. Если имя составлено из нескольких слов, то слова разделяют знаком подчеркивания (пробел не может быть частью имени):

```
#define SOME_LONG_MACRO_NAME
```

Макроподстановка-функция позволяет передать параметры (как фрагменты текста). Рекурсия не поддерживается. В качестве иллюстрации приведем следующий забавный пример (попробуйте его запустить).

Пример 18.2. Макроподстановка-функция

```
#define ONE(x) x
#define TWO(x) ONE(x) ONE(x)
#define FOUR(x) TWO(x) TWO(x)

#include <iostream>
int main() {
  FOUR(std::cout << "aaa\n";)
}
```

Параметров может быть несколько. В этом случае они разделяются запятыми. Соответственно, текст параметра не может содержать запятую (если это не строковое значение в кавычках).

Скобки влияют на связывание макроса: макроподстановка с параметрами не применяется, если после ее имени не идет открывающая скобка:

```
#define SEL(x, y) x

char (SEL)(char x, char y) {
  return y;
}

int main() {
  std::cout << SEL('a', 'b');     // макрос: a
  std::cout << (SEL)('a', 'b'); // функция: b
}
```

Чтобы препроцессор не воспринимал перечисление через запятую как набор параметров макроса, все перечисление можно взять в скобки:

```
#define A(x) x
A((1, 2)) // x == (1, 2)
```

После списка именованных параметров (или вместо него) в макроподстановке можно поставить многоточие. В этом случае можно передать произвольное число аргументов, все «лишние» аргументы будут доступны через имя __VA_ARGS__ в форме списка через запятую:

```
#define printf(...) fprintf(stdout, __VA_ARGS__)
```

undef (сокр. от *undefine*) — отменяет имеющееся определение макроподстановки.

```
#undef BEGIN
#undef END
// Теперь имена BEGIN и END снова свободны.
```

if — включение фрагмента кода по условию. Простейший вариант имеет следующий вид:

```
#if условие
   // Текст, включаемый, если условие истинно.
#endif
```

Доступен также вариант с перебором нескольких условий:

```
#if условие-1
   // Текст, включаемый, если условие-1 истинно.
#elif условие-2
   // Текст, включаемый, если условие-2 истинно,
   // а условие-1 ложно.
// Аналогичные ветки elif.
// Опциональная альтернативная ветка:
#else
   // Текст, включаемый, если все условия выше ложны.
#endif
```

Условие может быть выражением на C++, использующим определенные имена (макроподстановок) и целочисленные константы. Чтобы проверить, определена ли макроподстановка с некоторым именем, можно использовать запись defined(имя).

Для проверки доступности заголовочного файла, начиная с C++17, можно использовать запись __has_include(*аргумент-include*):

```
#if __has_include(<windows.h>)
#  define USE_WINDOWS
#  define WIN32_LEAN_AND_MEAN
#  include <windows.h>
#elif __has_include(<unistd.h>)
#  define USE_POSIX
#  include <unistd.h>
#endif
```

ifdef — сокращение для комбинации #**if** defined («если определено»).

ifndef — сокращение для комбинации #**if** !defined («если не определено»).

Типичный пример использования ifndef — стражи включения (см. с. 124).

Стандарт C++ определяет ряд макросов. Кроме того, компиляторы могут добавлять свои определения. Все они вместе называются *предопределенными макросами* [*predefined macros*], поскольку они определены уже на первой строчке программы.

Некоторые стандартные предопределенные макросы:

- __cplusplus — определен, если компилятор поддерживает C++. Разворачивается в числовую константу, зависящую от поддерживаемого стандарта. В случае C++17 это 201703L.
- __FILE__ заменяется именем транслируемого файла.
- __LINE__ заменяется числом — строчкой в исходном файле. Пример применения __FILE__ и __LINE__ приведен в конце раздела.
- __DATE__ и __TIME__ заменяются текущей датой и временем.

Некоторые другие часто встречающиеся предопределенные макросы:

- NDEBUG — обычно определяется при трансляции финальной версии, не предназначенной для отладки.
- _DEBUG — может быть определен в некоторых средах при трансляции отладочной версии.
- _WIN32 или WIN32 — обычно определяются при компиляции под ОС Windows.
- _MSC_VER — версия Microsoft (Visual) C++. Определен, если используется компилятор Microsoft.
- _DLL — определен компилятором Microsoft при сборке динамически связываемой библиотеки (DLL).

- __GNUC__ и __GNUC_MINOR__ — номер версии компилятора GCC. Определен, если используется компилятор GCC.

- __unix__ — определен при компиляции для ряда ОС семейства Unix (но не macOS).

- __APPLE__ — определен при компиляции для ОС компании Apple.

- __linux__ — определен при компиляции для ОС на основе GNU/Linux.

error — выводит сообщение об ошибке. Сообщение указывается в одну строку после #**error**.

line — подменяет текущий номер строки (что видно, например, при выводе ошибок). Вторым параметром можно указать новое имя текущего файла.

pragma[142] — иные директивы. Не стандартизованы.

У Написать макроподстановку-функцию DEBUG(msg), которая ведет себя по-разному, в зависимости от того, определена ли NDEBUG:

- Если определена, то подставлять пустую строку.

- Если не определена, то подставлять (std :: cout << (msg)).

Проверить, как ведет себя эта макроподстановка в *отладочной сборке* [*debug build*] и *финальной сборке*[143] [*release build*]. Способ управления сборкой определяется выбранной IDE (или напрямую компилятором).

При определении макроподстановок с параметрами в теле подстановки допустимо использовать особые операторы # (унарный) и ## (бинарный).

[142] От англ. *pragmatic*, др.-греч. *πραγματικός* — «относящийся к делу, дельный, подходящий», далее родств. слову *практика*.

[143] На жаргоне *релиз* от англ. *release* «выпуск».

Оператор # возвращает представление аргумента в виде строковой константы (т. е. то, что видит препроцессор, поскольку для препроцессора все аргументы суть просто последовательности символов, фрагменты файла исходного кода):

```
#define STR(x) #x
cout << STR(0xZZZ); // 0xZZZ
// → cout « "0xZZZ";
```

С помощью # также можно превратить в строковую константу список аргументов, доступный через __VA_ARGS__:

```
#define STR(...) #__VA_ARGS__
cout << STR(a, b, 0x, 3D);
// → "a, b, 0x, 3D"
```

Оператор ## выполняет конкатенацию аргументов (но не конвертирует их в строковый литерал). Сравните:

```
#define SAB(x, y) (#x #y)
#define LAB(x, y) x##y
cout << SAB(0x, F) << '\n';
     // → "0xF"→ "0xF"
cout << LAB(0x, F) << '\n';
     // → 0xF, выводится как 15
```

В качестве примера приведем реализацию макроса, напоминающего стандартный assert, но в случае ложности аргумента не прерывающего исполнение программы, а только выводящего сообщение об ошибке:

```
// Включен в отладочном режиме.
#ifndef NDEBUG
// Для отложенной подстановки:
#define STRINGIZE(x) STRINGIZE2(x)
#define STRINGIZE2(x) #x
// Символ \ в конце строки работает как перенос:
#define CHECK(x) \
  (void)(x? std::cerr: std::cerr << \
    (__FILE__ ",␣" STRINGIZE(__LINE__) \
    "␣assertion␣failed:␣" #x "\n"))
#else
```

```
// В финальной сборке не порождает кода.
#define CHECK(x)
#endif

#include <iostream>
int main() {
  CHECK(true);   // ничего не выведет
  CHECK(false); // выведет сообщение
  return 0;
}
```

18.3. Классы сложности алгоритмов

Одну и ту же задачу можно решать различными способами. В частности, некоторую вычислимую функцию обычно можно реализовать множеством способов даже в рамках одного вида вычислителей. Каждая такая реализация может считаться самостоятельной программой, являющейся воплощением некоторого алгоритма.

Итак, одному алгоритму может соответствовать множество его воплощений в виде конкретных программ, которые можно сравнивать между собой с разных точек зрения: краткости, эффективности использования аппаратных ресурсов, надежности и т. д. Но на более высоком уровне абстракции также нет единства: вычислять одну и ту же функцию (т. е. решать одну и ту же задачу) можно с помощью разных алгоритмов. Эти алгоритмы опять же можно сравнивать с разных точек зрения: простоты реализации (перевода в программы), количества действий, которые требуется выполнить исполнителю, объема памяти, занимаемого промежуточными данными, полноты.

Под полнотой здесь понимается степень покрытия различных возможных ситуаций. Строго говоря, формально разная полнота соответствует разным задачам, но на практике часто бывает удобно рассматривать такие задачи как одну. Простой пример: задача решить линейное уравнение. Мы можем зара-

нее предполагать (или точно знать), что уравнение имеет решение — одна задача. А можем не знать, имеет ли уравнение решение — другая задача, алгоритм решения которой включает проверку существования решения.

Под **сложностью алгоритма** [*algorithm complexity*] понимается зависимость количества операций абстрактного исполнителя (условного компьютера) от размера исходной задачи. Размер задачи необязательно прямо пропорционален размеру ввода (описания задачи). Например, задача «найти количество простых чисел, меньших N» имеет размер N, в то время как для описания ввода достаточно $\log_2 N$ бит.

Пусть даны две функции f и g, отображающие $\mathbb{N}$ в $\mathbb{N}$ (для простоты часто расширяют до действительных чисел, например, указывают логарифм или корень без округления).

Тогда говорят, что f лежит в классе:

- $\omega(g(n))$, если $\forall C > 0\ \exists n_0 \colon n > n_0 \Rightarrow Cg(n) < f(n)$ — функция $f(n)$ растет асимптотически быстрее, чем $Cg(n)$ для любой положительной константы C;
- $o(g(n))$, если $\forall C > 0\ \exists n_0 \colon n > n_0 \Rightarrow f(n) < Cg(n)$ — функция $f(n)$ растет асимптотически медленнее, чем $Cg(n)$ для любой положительной константы C;
- $\Omega(g(n))$, если $\exists C > 0\ \exists n_0 \colon n > n_0 \Rightarrow Cg(n) \leqslant f(n)$ — функция $f(n)$ растет асимптотически не медленнее, чем $Cg(n)$ для некоторой положительной константы C;
- $O(g(n))$, если $\exists C > 0\ \exists n_0 \colon n > n_0 \Rightarrow f(n) \leqslant Cg(n)$ — функция $f(n)$ растет асимптотически не быстрее, чем $Cg(n)$ для некоторой положительной константы C;
- $\Theta(g(n))$, если $\exists C_1 > 0\ \exists C_2 \geqslant C_1\ \exists n_0 \colon n > n_0 \Rightarrow C_1 g(n) \leqslant \leqslant f(n) \leqslant C_2 g(n)$ — скорость роста функций $f(n)$ и $g(n)$ асимптотически одинакова с точностью до положительной константы.

Класс $\Theta(g(n))$ является пересечением $\Omega(g(n))$ и $O(g(n))$.

Использование данных обозначений позволяет абстрагироваться от конкретных числовых оценок сложности отдельных элементарных операций (TIME, подразумевается, если не указано иначе) или количества ячеек памяти, занимаемых отдельными значениями базовых элементарных типов (SPACE).

Например, линейный поиск в массиве требует $O(n)$ действий, где n — размер массива (размер задачи). В то же время двоичный поиск в упорядоченном массиве размера n требует $O(\log_2 n)$ действий. Так как логарифмы одного аргумента по разным основаниям отличаются друг от друга множителем-константой, то обычно пишут $O(\log n)$ без указания основания. Оба алгоритма поиска не требуют для работы дополнительной памяти, что обозначается через $O(1)$-SPACE или короче: SPACE(1) — объем требуемой памяти не зависит от размера задачи.

Строго говоря, представление некоторого идентификатора элемента множества размера n требует $O(\log n)$ символов, поэтому даже для поиска запись $O(1)$-SPACE является упрощением, связанным с тем, что используемые на практике идентификаторы (индексы, адреса) нередко либо имеют ограниченный сверху размер, либо их размер не связан непосредственно с размером задачи (например, почтовые адреса).

Если для некоторой задачи лучшие алгоритмы ее решения попадают в класс сложности $O(g(n))$, то говорят, что задача имеет сложность $O(g(n))$.

Какой вариант поиска быстрее?[144] Поскольку $n \in \omega(\log n)$, то какова бы ни была корректная реализация двоичного и линейного поисков, всегда найдется n, начиная с которого двоичный поиск будет работать быстрее. С другой стороны, это n может оказаться не таким уж маленьким. Если, например, приходится постоянно искать значения в массивах из нескольких элементов, то линейный поиск почти наверняка будет ра-

[144] Здесь мы закрыли глаза на тот факт, что линейный и двоичный поиск имеют разные предусловия: для корректной работы двоичного поиска требуется, чтобы массив был отсортирован.

ботать быстрее двоичного. Это довольно типичная ситуация в мире алгоритмов: асимптотически более медленные алгоритмы обычно проще (в реализации) более быстрых и, как правило, выигрывают в скорости на реальных задачах малого размера. Поэтому их использование может быть оправдано на практике.

Итак, $\Theta(\log n)$-TIME алгоритмы называют *логарифмическими*, $\Theta(n)$-TIME — *линейными* [*linear time*], $\Theta(n^2)$-TIME — *квадратичными*, $\Theta(n^3)$-TIME — *кубическими* и т. д. $\Theta(n \log n)$-TIME алгоритмы называют *линеарифмическими* [*linearithmic time*]. Вместо $\Theta(\cdot)$ часто используется оценка сверху $O(\cdot)$, если она достигается (оценка по *худшему случаю*).

В таблице ниже дано сравнение разных функций роста сложности. Для удобства восприятия все числа даны в единицах времени (округлены, знак * означает «не менее миллиарда лет»): с — секунды, м — минуты, ч — часы, д — дни, г (л) — годы, тл — тысячи лет, мл — миллионы лет.

n	2 с	4 с	8 с	1 м	2 м	1 ч	2 ч	1 д
$\log_2 n$	1 с	2 с	3 с	6 с	7 с	12 с	13 с	17 с
$\log_2^2 n$	1 с	4 с	9 с	35 с	48 с	140 с	165 с	4 м
$\sqrt{n}$	2 с	2 с	3 с	8 с	11 с	60 с	85 с	5 м
$n \log_2 n$	2 с	8 с	24 с	6 м	14 м	12 ч	26 ч	16 д
$n\sqrt{n}$	3 с	8 с	23 с	8 м	22 м	60 ч	7 д	294 д
n^2	4 с	8 с	64 с	1 ч	4 ч	150 д	1.7 г	237 л
$n^2 \log_2 n$	4 с	32 с	3 м	6 ч	28 ч	5 л	21 г	3879 л
$n^2\sqrt{n}$	6 с	32 с	3 м	8 ч	2 д	25 л	139 л	70 тл
n^3	8 с	64 с	9 м	2.5 д	20 д	1.5 тл	12 тл	20 мл
n^4	16 с	4 м	1 ч	150 д	7 л	5.4 мл	85 мл	*
2^n	4 с	16 с	4 м	*	*	*	*	*
$2^n n$	8 с	64 с	34 м	*	*	*	*	*
$(n-1)!$	1 с	6 с	1.4 ч	*	*	*	*	*
$n!$	2 с	24 с	11 ч	*	*	*	*	*
n^n	4 с	4 м	194 д	*	*	*	*	*

Вторая по скорости роста из приведенных в таблице функция $n!$ соответствует перебору всех возможных перестановок из n объектов и поначалу растет вроде бы не так уж и быстро, опережая, например, и n^3, и $2^n n$ при $n > 5$. Затем она резко «идет на взлет». Поэтому для малых n (1–4) алгоритмы,

перебирающие все перестановки, вполне могут быть пригодны на практике и даже более эффективны, чем другие алгоритмы ввиду своей простоты. Но если n велико, то никакой обозримый и физически реалистичный прогресс вычислительной техники не спасет ситуацию. Например, $30!$ с на порядки превосходит предполагаемый возраст Вселенной и даже $30! \cdot 10^{-18}$ с ≈ 8.4 млн лет. Если же в пространстве перестановок можно организовать двоичный поиск, то получим $\log n! \in O(n \log n)$ — линеарифмическую сложность. В этот класс попадает, в частности, задача сортировки последовательности объектов, на которых задано отношение линейного порядка.

Рассмотрим задачи вычисления предикатов.

Формальные языки, понимаемые как произвольные множества строк, также относят к тем или иным классам сложности: если задача «определить, принадлежит ли строка языку L» попадает в некоторый класс сложности C, то говорят, что $L \in C$. Размер такой задачи есть длина проверяемой строки.

L или **LSPACE** — класс сложности языков $O(\log n)$-SPACE. Для решения задач из этого класса размера n требуется логарифмический по n объем памяти. Примеры: сравнение натуральных чисел произвольной величины, вычисление значения фиксированной булевской формулы для заданных значений переменных.

P или **PTIME** — *полиномиальное время*, объединение классов $O(n^k)$-TIME для всех $k \in \mathbb{N}$. Примеры: проверка натурального числа на простоту, проверка системы линейных неравенств на разрешимость, большинство языков программирования (т. е. задачи проверки синтаксической корректности программ). Можно показать, что LSPACE $\subseteq$ P, так как при решении задачи из класса LSPACE количество всех возможных состояний (строк алфавита $\mathcal{A}$) в случае полного перебора принадлежит $O(|\mathcal{A}|^{C \log_2 n}) = O(n^{C \log_2 |\mathcal{A}|})$. На данный момент неизвестно, истинно ли утверждение «LSPACE = P».

PSPACE — объединение классов $O(n^k)$-SPACE для всех $k \in \mathbb{N}$. Для решения задачи требуется полиномиальный объем памя-

ти. Пример: вычисление булевской формулы, в которой разрешены кванторы существования и всеобщности для переменных. Очевидно, что P $\subseteq$ PSPACE. Неизвестно, является ли утверждение «P = PSPACE» истинным.

EXP или **EXPTIME** — *экспоненциальное время*, объединение классов $O(2^{n^k})$-TIME для всех $k \in \mathbb{N}$. Имеем P $\subset$ EXP, PSPACE $\subseteq$ EXP. Неизвестно, является ли истинным утверждение «PSPACE = EXP».

EXPSPACE — объединение классов $O(2^{n^k})$-SPACE для всех $k \in \mathbb{N}$. Верны утверждения: PSPACE $\subset$ EXPSPACE, EXPTIME $\subseteq$ EXPSPACE. Неизвестно, истинно ли утверждение «EXPTIME = EXPSPACE».

Конечно, можно ввести еще более трудоемкие классы, однако с практической точки зрения уже задачи из PSPACE считаются почти нерешаемыми.

В случае, если решение задачи представляет собой нечто более сложное, нежели ответ «да-нет», вводятся аналогичные классы, к названиям которых добавляется буква **F** (*функциональный*): например, класс FP содержит арифметические операции над рациональными числами, а также задачи приближенного вычисления значений многих элементарных и специальных функций.

Задачи вычисления отрицания предиката (дополнения) могут порождать собственные классы сложности, обозначаемые с помощью приставки **co-**[145]. Для всех вышеперечисленных классов задач вычисления предикатов аналогичные co-классы, очевидно, совпадают с исходными: L = co-L, P = co-P и т. д., так как достаточно решить «прямую» задачу и выполнить булевское отрицание.

Различие возникает при введении *недетерминированных автоматов* [*non-deterministic automata*] и соответствующих им классов сложности. Недетерминированный автомат является абстрактным автоматом, который может одновременно нахо-

[145] От лат. *complementum* — «дополнение».

диться во множестве состояний. Можно считать, что недетерминированный автомат строит *множество достижимости* некоторого детерминированного автомата.

Соответствующие классы сложности обозначаются с помощью приставки **N**: NL, NP, NEXP. Известно, что

$$\mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXP} \subseteq \mathrm{NEXP} \subseteq \mathrm{EXPSPACE}.$$

Где-то в этом ряду должны быть строгие включения, так как $\mathrm{P} \subset \mathrm{EXP}$, $\mathrm{NP} \subset \mathrm{NEXP}$ и $\mathrm{L} \subset \mathrm{PSPACE} \subset \mathrm{EXPSPACE}$, а также $\mathrm{NL} = \text{co-NL}$, $\mathrm{PSPACE} = \mathrm{NPSPACE}$, $\mathrm{EXPSPACE} = \mathrm{NEXPSPACE}$. Неизвестно, верны ли утверждения: «$\mathrm{L} = \mathrm{NL}$», «$\mathrm{NL} = \mathrm{P}$», «$\mathrm{P} = \mathrm{NP}$», «$\mathrm{NP} = \mathrm{PSPACE}$», «$\mathrm{NP} = \text{co-NP}$», «$\mathrm{EXP} = \mathrm{NEXP}$», «$\mathrm{NEXP} = \text{co-NEXP}$». Доказательство или опровержение равенства $\mathrm{P} = \mathrm{NP}$ является наиболее широко известной открытой теоретической проблемой в теории алгоритмов.

В терминах детерминированных автоматов задачи разрешимости из N-классов формулируются как задачи проверки *сертификата*, т. е. своего рода доказательства ответа «да» или «нет». Так, задача вычисления предиката $P(x)$ принадлежит NP, если существует задача вычисления предиката $R(x, y)$ из P, такого, что для любого x найдется y (сертификат), при котором $R(x, y) = P(x)$. Фактически сертификат — это и есть решение по существу.

Пример. Пусть даны набор чисел размера N и некоторое число a. Задача: существует ли подмножество этого набора чисел такое, что его сумма равна a? Сертификат: само такое подмножество. Для проверки корректности сертификата требуется сравнить сумму всех чисел с a и проверить принадлежность каждого числа исходному набору (что можно сделать, например, линейным поиском). Итак, данная задача относится к классу NP.

Говорят, что задача X **полиномиально сводима** к задаче Y, если существует алгоритм решения X, использующий решение задачи Y, полиномиальное по размеру ввода число раз.

Если любая задача из класса C полиномиально сводима к задаче X, то X называют C-трудной [*C-hard*] задачей. Если X, являясь C-трудной, сама принадлежит классу C, то ее называют C-полной [*C-complete*] задачей.

Пример (задача коммивояжера). Пусть дан набор городов и расстояний между ними, а также число d. Требуется узнать, существует ли маршрут, проходящий по каждому городу ровно один раз и при этом имеющий длину не более d.

Легко показать, что эта задача принадлежит классу NP. Что более интересно, так это тот факт, что задача также является NP-трудной (а значит, NP-полной).

У Покажите, что задача коммивояжера принадлежит NP.

Известен целый ряд NP-полных задач. Если для какой-либо из них обнаружится полиномиальный алгоритм решения, то P = NP.

Задача нахождения проходящего через все города маршрута минимальной длины (собственно классическая задача коммивояжера) принадлежит классу FNP. На практике подобные задачи требуют экспоненциального времени для нахождения точного решения. В то же время обычно известны методы сравнительно быстрого построения относительно хорошего, но возможно неоптимального решения (*субоптимального решения*).

У Ответьте на следующие вопросы:

- Пусть $n \in \mathbb{N}$ — размер задачи. Что лучше: алгоритм, требующий $\left\lceil 10 + \left(\frac{n}{25}\right)^{10} \right\rceil$ операций, или алгоритм, требующий $\lceil 100 + 1.1^n \rceil$ операций?

- Какому классу сложности принадлежит задача проверки того, является ли произвольная последовательность n натуральных чисел упорядоченной по возрастанию?

- Какому классу сложности принадлежит задача, решение которой требует $O(n!)$ операций?

- Рассмотрим следующую задачу (о составлении латинского квадрата). Пусть $n \in \mathbb{N}$. Пусть дан алфавит $\mathcal{A}$, $|\mathcal{A}| = n$.

Пусть дана $n \times n$-матрица, каждая из ячеек которой может быть пуста или содержать символ алфавита $\mathcal{A}$. При этом заполненные ячейки в каждом столбце, в каждой строке и в каждой диагонали матрицы должны содержать разные символы. К какому классу сложности относится задача определения возможности дозаполнить заданную матрицу символами алфавита $\mathcal{A}$, не нарушая указанного условия?

18.4. Алгоритмы сортировок

C++ позволяет сделать тип или константу времени компиляции параметром определения. Такое определение называется *шаблоном* [*template*]. При его использовании компилятор подставляет конкретные типы-параметры и константы времени компиляции и порождает новое определение для каждого различного случая.

Для определения шаблона используется ключевое слово **template**, которое ставится перед определением и после которого в «угловых скобках» (символы «меньше» и «больше») указывается список параметров шаблона. Параметр-тип указывается с помощью ключевого слова **class** или **typename** (семантической разницы нет).

Например, простая функция:

```
// Возводит в квадрат любое число.
template <class Num>
Num sqr(Num x) {
  return x*x;
}
```

Данное определение шаблона применимо к любому типу Num такому, что при подстановке его вместо Num получаем корректное определение функции, т. е. определен оператор $*$ и его результат совместим с Num.

При применении шаблона можно указать значения аргументов после имени шаблона аналогично в угловых скобках.

```
cout << sqr<int>(12) << '\n';
cout << sqr<int>(1.2); // выведет 1
```

В случае шаблонов функций (и типов, если их имена используются как конструкторы) аргументы-типы можно не указывать, если компилятор способен их определить по типам аргументов вызова функции. В частности, в примере выше первый вызов sqr и так получает аргумент типа **int**, поэтому <**int**> можно было не писать:

```
// То же самое, выведет 144:
cout << sqr(12) << '\n';
```

Ниже мы рассмотрим ряд алгоритмов сортировки и приведем их реализации в виде шаблонов функций, которые принимают указатели произвольного типа. Предполагается, что оператор < задает линейный порядок на типе сортируемых значений. Сортировать будем элементы в массивах. Сортируемый полуинтервал в массиве будем передавать парой указателей (на первый элемент и на элемент за последним, разность этих указателей равна количеству элементов).

18.4.1. Квадратичные алгоритмы

Сортировки на основе сравнения элементов предполагают выполнение двух основных операций: сравнение на «меньше» и присваивание (или обмен значений). Из практических соображений иногда можно использовать (как особую операцию) операцию обмена [*swap*]. Мы будем использовать стандартную функцию std::swap (заголовочный файл **utility**).

Здесь мы не будем рассматривать самый известный (и один из самых медленных) алгоритм сортировки — сортировку пузырьком, сосредоточившись на практичных алгоритмах.

Сортировка выбором одного элемента

Алгоритм очень прост:

1. Найдем в диапазоне минимум и обменяем его с первым элементом диапазона.

2. Отрежем от диапазона первый элемент.

3. Будем повторять этот процесс, пока диапазон не станет меньше двух элементов.

Пример 18.3. Функция поиска минимума на диапазоне

```
template <class T>
T * select_min(T * from, T * to) {
  T * found = from;
  if (from != to) while (++from != to)
    if (*from < *found)
      found = from;
  return found;
}
```

Алгоритм можно сформулировать рекурсивно:

1. Если диапазон меньше двух элементов, то он уже отсортирован.

2. Обменяем минимум диапазона с первым элементом.

3. Отсортируем хвост (диапазон без первого элемента).

Пример 18.4. Сортировка выбором минимума, рекурсия

```
template <class T>
void select_min_sort(T * from, T * to) {
  if (to - from < 2) return;
  std::swap(*from, *select_min(from, to));
  return sort_select(from + 1, to);
}
```

Конечно, вариант с рекурсией не следует использовать в C++. Здесь от рекурсии очень просто избавиться — имеем чистый хвостовой вызов. Кроме того, нет смысла выполнять обмен, если минимум уже лежит в начале диапазона — можно добавить соответствующую проверку.

Пример 18.5. Сортировка выбором минимума, цикл

```
template <class T>
void select_min_sort(T * from, T * to) {
  for (; 1 < to - from; ++from)
    if (T * m = select_min(from, to); m != from)
      std::swap(*from, *m);
}
```

Сортировка выбором двух элементов

Очевидно, что выше мы могли точно так же выбирать максимум вместо минимума. Но полезнее сразу за один проход по диапазону находить и минимум, и максимум, уменьшая диапазон на два элемента за один проход. Впрочем, данный метод намного сложнее, потому что надо правильно обрабатывать ряд неочевидных случаев.

Сначала напишем функцию, находящую сразу минимум и максимум. Пусть она возвращает пару указателей (первый — на минимум, второй — на максимум).

Пример 18.6. Функция поиска минимума и максимума

```
template <class T>
std::pair<T*, T*> select_min_max(T * from, T * to) {
  T * cur_min = from;
  T * cur_max = from;
  if (from != to) while (++from != to)
    if (*from < *cur_min)
      cur_min = from;
    else if (!(*from < *cur_max))
      cur_max = from;
  return { cur_min, cur_max };
}
```

Можно заметить, что если минимум и максимум совпали, то сортировку можно заканчивать. Если диапазон содержит меньше двух элементов, то минимум и максимум (указатели на них) совпадут.

Пример 18.7. Сортировка выбором минимума и максимума

```
template <class T>
void select_min_max_sort(T * from, T * to) {
  for (;;) { // ввести переменные min, max как части пары:
    auto [min, max] = select_min_max(from, to);
    if (min == max) return;
    --to; // теперь указатель на последний
    if (max != from) {
      if (min != from) std::swap(*min, *from);
      if (max != to)   std::swap(*max, *to);
    } else {
      if (max != to)   std::swap(*max, *to);
      if (min != to)   std::swap(*min, *from);
    }
    from += from != to;
  }
}
```

Итак, сортировка выбором производит квадратичное число сравнений и не более, чем линейное число обменов. Если последовательность уже отсортирована, то сортировка выбором не выполнит ни одного обмена (присваивания).

Сортировка вставками

Сортировка вставками основана на принципе спуска каждого элемента («камня») вниз до надлежащей позиции с вытеснением прочих элементов вверх. Иными словами, диапазон разделяется на две части: уже отсортированную («низ»), вначале состоящую из первого элемента, и еще не отсортированную («верх»). Далее мы на каждой итерации вставляем нижний (первый) элемент неотсортированной части в отсортированную часть. Так как вставка в массив требует сдвига части элементов, мы совмещаем сдвиг элементов с поиском позиции вставки.

Формально этот алгоритм весьма прост, но наивная реализация на каждой итерации выполняет пару лишних присваиваний, если очередной элемент оказывается не меньше всех уже

отсортированных (т. е. его некуда смещать вниз). Кроме того, по возможности мы выполняем перемещение (move), а не копирование значений.

Пример 18.8. Сортировка вставками

```
// Вставка элемента *to в [from, to)
//    (отсортированную часть).
template <class T>
void sorted_augment(T * from, T * to) {
  assert(from != to);
  T * prev = to - 1;
  if (*prev < *to) return;
  // Вытащим «камень» из массива.
  T stone = std::move(*to);
  // Освобождение позиции для «камня».
  *to = std::move(*prev);
  while (from != prev) {
    T * pprev = prev - 1;
    if (!(stone < *pprev)) break;
    *prev = std::move(*pprev);
    prev = pprev;
  }
  // Поставим «камень» в массив.
  *prev = std::move(stone);
}

// Сортировка вставками.
template <class T>
void insert_sort(T * from, T * to) {
  if (from == to) return;
  for (T * bound = from; ++bound != to;)
    sorted_augment(from, bound);
}
```

Сортировка вставками может производить квадратичное число и сравнений, и присваиваний. Однако, если последовательность уже отсортирована, то присваиваний можно избежать, а сравнений будет линейное число.

Если последовательность близка к отсортированной, то скорость сортировки вставками может приближаться к линейной. Благодаря этому свойству данный алгоритм можно использовать как завершающий этап более сложных алгоритмов, быстро приводящих последовательность к «почти отсортированной».

Если последовательность содержит небольшое количество элементов (менее 30), то сортировка вставками может оказаться быстрейшим алгоритмом среди универсальных, поэтому ее используют как часть гибридных алгоритмов сортировки для сортировки коротких диапазонов.

Эквивалентными элементы a и b называют, если истинно $\neg(a < b) \wedge \neg(b < a)$, т. е. ни один из них не меньше другого.

Если элементы равны, то они эквивалентны. Но возможны ситуации, когда для сравнения используется лишь часть представления значений, в этом случае эквивалентные элементы могут быть не равны. Другой пример не равных, но эквивалентных элементов — нечисла по стандарту IEEE-754.

Устойчивой сортировкой [*stable sort*] называют сортировку, сохраняющую исходный относительный порядок эквивалентных элементов.

Устойчивость является полезным свойством. Если сортировка устойчива, то можно комбинировать сортировки по разным критериям. Например, отсортировать файлы сначала по дате, потом по типу, потом по имени.

Представленная выше реализация сортировки вставками является устойчивой.

18.4.2. Линеарифмические алгоритмы

Сортировка слиянием

Простейшим для понимания и в то же время весьма эффективным на практике является алгоритм сортировки слиянием[146].

Данный алгоритм определяется рекурсивно:

1. Если диапазон содержит менее двух элементов, то закончить.

2. Разделить диапазон пополам и отсортировать каждую половину.

3. Выполнить *слияние* двух отсортированных половин, чтобы получить сортировку последовательности целиком.

Слияние [*merge*] — линейная по числу действий процедура, которая объединяет две (или более) отсортированных последовательностей в одну отсортированную последовательность, содержащую все элементы исходных последовательностей.

Представим код процедуры слияния, принимающей два исходных диапазона и адрес начала результирующей последовательности (dest). Предполагается, что памяти по этому адресу достаточно для записи всех исходных элементов.

Пример 18.9. Процедура слияния

```
// Перемещает один диапазон в другой.
template <class T>
T * move(T const * from, T const * to, T * dest) {
  while (from != to)
    *dest++ = std::move(*from++);
  return dest;
}
```

[146] По утверждению Д. Кнута, данный алгоритм был предложен Дж. фон Нейманом в 1945 г.

```
// Возвращает адрес элемента
//    после последнего записанного.
template <class T>
T * merge(
    T const * a_from, T const * a_to,
    T const * b_from, T const * b_to,
    T * dest) {
  for (;;) {
    // Цикл перемещения из a.
    for (;;) {
      if (a_from == a_to)
        return move(b_from, b_to, dest);
      if (*b_from < *a_from)
        break;
      *dest++ = std::move(*a_from++);
    }
    // Цикл перемещения из b.
    for (;;) {
      if (b_from == b_to)
        return move(a_from, a_to, dest);
      if (!(*b_from < *a_from))
        break;
      *dest++ = std::move(*b_from++);
    }
  }
}
```

Используя функцию слияния, не составит труда записать рекурсивную сортировку слиянием.

Пример 18.10. Сортировка слиянием

```
// Использует внешний буфер
//    размера исходной последовательности.
template <class T>
void merge_sort(T * from, T * to, T * buf) {
  if (to - from < 2) return;
  T * const mid = from + (to - from)/2;
  merge_sort(from, mid, buf);
  merge_sort(mid, to, buf);
  move(buf,
```

```
        merge(from, mid, mid, to, buf),
        from);
}

// Создает буфер нужного размера.
template <class T>
void merge_sort(T * from, T * to) {
  auto buf = std::make_unique<T[]>(to - from);
  merge_sort(from, to, buf.get());
}
```

Данную реализацию можно несколько улучшить, если заметить, что исходный массив также можно использовать как буфер — можно избавиться от перемещений на каждом вызове merge_sort, выполнив решение в виде взаимной рекурсии двух функций: merge_sort, сохраняющей результат в исходном массиве, и merge_sort_retro, сохраняющей результат в буфере. Другой оптимизацией может быть вызов insert_sort, если длина диапазона достаточно мала (здесь для простоты выбрана константа, не зависящая от типа сортируемых элементов).

Пример 18.11. Сортировка слиянием, челночный гибрид

```
// Прототип.
template <class T>
void merge_sort_retro(T * from, T * to, T * buf);

template <class T>
void merge_sort(T * from, T * to, T * buf) {
  auto const span = to - from;
  if (span < 30)
    return insert_sort(from, to);
  auto const half = span/2;
  T * const mid = from + half;
  T * const buf_mid = buf + half;
  merge_sort_retro(from, mid, buf);
  merge_sort_retro(mid, to, buf_mid);
  merge(buf, buf_mid, buf_mid, buf + span, from);
}
```

```
template <class T>
void merge_sort_retro(T * from, T * to, T * buf) {
  auto const span = to - from;
  if (span < 30)
    return insert_sort(buf, move(from, to, buf));
  auto const half = span/2;
  T * const mid = from + half;
  T * const buf_mid = buf + half;
  merge_sort(from, mid, buf);
  merge_sort(mid, to, buf_mid);
  merge(from, mid, mid, to, buf);
}
```

Избавляться от рекурсии в случае сортировки слиянием не имеет практического смысла. На каждом шаге мы делим диапазон пополам, поэтому глубина рекурсии не превосходит $\lceil \log_2 N \rceil$, где N — количество элементов, что позволяет легко показать, что сама сортировка требует порядка $N \log N$ операций сравнения и перемещения. К сожалению, количество выполняемых операций почти не зависит от того, близка исходная последовательность к отсортированной или нет. Впрочем, когда требуется одинаковое время исполнения операции, не зависящее от исходных данных[147], это может стать преимуществом.

Сортировка слиянием устойчива. Основной ее недостаток — необходимость использовать вспомогательную память объема N (буфер).

Пирамидальная сортировка

Естественно, может возникнуть вопрос, существует ли линеарифмическая сортировка, не требующая буфера. Ответ на него положительный. Алгоритм (*пирамидальная сортировка*

[147] Некоторые атаки на криптосистемы базируются на оценке времени, затрачиваемого на обработку запросов.

[*heap sort*]), обладающий такими свойствами, был предложен[148] в 1964 г.

Пирамидальная сортировка основана на введении особой структуры данных — *пирамиды* (*кучи* [*heap*]). Пирамида позволяет выполнять вставку элемента и извлечение максимума (и/или минимума). Для сортировки этих операций достаточно: вначале мы заполняем пирамиду всеми элементами последовательности, затем повторяем извлечение максимального элемента, пока пирамида не опустеет.

Пока не ясно, каким образом можно избежать затрат дополнительной памяти. Дело в том, что пирамиду можно построить прямо в исходном массиве (как упакованное двоичное дерево), причем за линейное время.

Ниже мы рассмотрим конкретные виды пирамид (коих придумано уже немалое множество): двоичную и троичную.

С формальной точки зрения пирамида является деревом: у каждого элемента (*узла*) может быть один *предок* и несколько (до двух в двоичной и до трех в троичной) *потомков*. Предки есть у всех узлов, кроме одного, называемого *корнем* или *вершиной пирамиды*. Узлы, не имеющие потомков, называются *листьями*.

Инварианты пирамиды, уложенной в массив:

1. Значения потомков узла не могут быть больше значения самого узла. Таким образом, на вершине лежит максимум.

2. Все уровни (кроме, может быть, последнего) заполнены полностью.

3. Последний уровень заполняется слева направо.

Укладка пирамиды в массиве имеет следующий вид: элемент 0 (первый) является корнем, потомки элемента i в дво-

[148] *Williams J.* Algorithm 232 — Heapsort // Comm. ACM. 1964. № 7(6). P. 347–348.

ичной куче суть элементы $2i+1$ и $2i+2$, в троичной куче — элементы $3i+1$, $3i+2$ и $3i+3$.

Строить эту укладку можно с конца массива, таким образом, максимум будет продвигаться в начало.

Пример 18.12. Создание пирамиды

```
// Корректировка поддерева с корнем i.
template <class T, class I>
void adjust_heap2(T * from, I sz, I i = I(0)) {
  for (T * root = from + i;;) {
    auto const
      c1 = from + (2*i + 1),
      c2 = from + (2*i + 2);

    T * max = root;
    if (c1 - from < sz && *max < *c1)
      max = c1;
    if (c2 - from < sz && *max < *c2)
      max = c2;
    if (max == root)
      return;

    std::swap(*root, *max);

    // Спуститься в измененное поддерево, чтобы
    root = max; // выполнить инварианты пирамиды.
    i = max - from;
  }
}

// Формирование пирамиды «на месте».
template <class T>
void make_heap2(T * from, T * to) {
  auto const sz = to - from;
  assert(sz >= 0);
  for (auto i = sz/2; 0 <= i; --i)
    adjust_heap2(from, sz, i);
}
```

Теперь сама сортировка записывается элементарно.

Пример 18.13. Пирамидальная сортировка

```
// Сортировка пирамиды, уложенной в [from, to).
template <class T>
void sort_heap2(T * from, T * to) {
  while (from != --to) {
    std::swap(*from, *to);
    adjust_heap2(from, to - from);
  }
}
// Пирамидальная сортировка.
template <class T>
void heap2_sort(T * from, T * to) {
  make_heap2(from, to);
  sort_heap2(from, to);
}
```

Данный код несложно модифицировать для использования троичной пирамиды.

Пример 18.14. Троичная пирамида

```
// Корректировка поддерева с корнем i.
template <class T, class I>
void adjust_heap3(T * from, I sz, I i = I(0)) {
  for (T * root = from + i;;) {
    auto const
      c1 = from + (3*i + 1),
      c2 = c1 + 1,
      c3 = c1 + 2;
    T * max = root;
    if (c1 - from < sz && *max < *c1)
      max = c1;
    if (c2 - from < sz && *max < *c2)
      max = c2;
    if (c3 - from < sz && *max < *c3)
      max = c3;
    if (max == root)
      return;
    std::swap(*root, *max);
```

```
      // Спуститься в измененное поддерево, чтобы
      root = max; // выполнить инварианты пирамиды.
      i = max - from;
    }
}

// Формирование пирамиды «на месте».
template <class T>
void make_heap3(T * from, T * to) {
  auto const sz = to - from;
  assert(sz >= 0);
  for (auto i = sz/3; 0 <= i; --i)
    adjust_heap3(from, sz, i);
}

// Далее заменить make_heap2 на make_heap3,
// adjust_heap2 на adjust_heap3.
```

Пирамидальная сортировка на основе троичной пирамиды выполняет несколько меньше операций сравнения и перемещения по сравнению с пирамидальной сортировкой на основе двоичной пирамиды.

Итак, пирамидальная сортировка на любых наборах данных требует линеарифмическое число сравнений и не менее линейного числа перемещений. Ей не требуется вспомогательная память.

Если последовательность уже отсортирована, то построение пирамиды вначале переставляет элементы, нарушая сортировку. Если последовательность отсортирована в обратном порядке, то она уже удовлетворяет требованиям пирамиды.

К сожалению, у данного алгоритма помимо неспособности быстро сортировать уже почти отсортированные массивы есть и иные недостатки. Во-первых, пирамидальная сортировка не является устойчивой. Во-вторых, на современных компьютерах пирамидальная сортировка демонстрирует сравнительно низкую скорость работы из-за неудачного использования кэшей процессоров: доступ осуществляется с непостоянным шагом.

Быстрая сортировка

Последний алгоритм, который мы рассмотрим в данном разделе, называется *быстрая сортировка* [*quick sort*]. Данный алгоритм был придуман Ч. Хоаром[149] во время его поездки в СССР в 1959 г. и стал, пожалуй, самым популярным из алгоритмов сортировки, поскольку в среднем обычно работает быстрее прочих сортировок на основе сравнений, вполне оправдывая свое название.

В некотором смысле, данный метод является противоположностью сортировки слиянием. Там мы сначала сортировали половины, потом объединяли результаты в единую сортировку. Здесь же мы вначале *разделяем* последовательность на часть, содержащую элементы меньше заданного, и часть, содержащую прочие элементы, а потом рекурсивно независимо сортируем получившиеся части. Благодаря такому маневру отпадает необходимость в буфере, так как разделение можно сделать на месте.

Простейший вариант данного алгоритма дан в виде листинга 18.15 (функция find используется в предусловии функции, выполняющей разделение, — данная функция предполагает наличие ключа в разделяемом диапазоне).

Пример 18.15. Наивная быстрая сортировка

```
template <class T>
T * find(T * from, T * to, T const & key) {
  while (from != to && *from != key)
    ++from;
  return from;
}

// Разделение элементов на две части по ключу,
//    принадлежащему диапазону.
template <class T>
T * key_part2_inplace(
```

[149] См.: *Hoare C. A. R.* Algorithm 64: Quicksort // Comm. ACM. 1961. № 4(7). P. 321.

```
    T * from, T * to,
    T const & key) {
  assert(from != to);
  assert(find(from, to, key) != to);
  T * l = from;
  T * r = to - 1;
  for (;;) {
    while (*l < key)
      ++l;
    while (key < *r)
      --r;
    if (r <= l)
      return r + (r != l);
    std::swap(*l, *r);
  }
}

template <class T>
void quick_sort(T * from, T * to) {
  if (to - from < 2) return;
  T const key = from[(to - from)/2];
  T * p = key_part2_inplace(from, to, key);
  quick_sort(from, p);
  quick_sort(p , to);
}
```

Устойчивость быстрой сортировки зависит от алгоритма, выполняющего разбиение. Вариант, приведенный выше, устойчивым не является.

В качестве ключа выбирается средний элемент, что позволяет данной версии эффективно работать, если исходная последовательность уже отсортирована в прямом или обратном порядке. Количество операций сильно зависит от того, попадает ли выбранный ключ близко к середине отсортированного диапазона или нет.

Если ключ выбирается неудачно, то на каждом шаге рекурсии происходит отделение лишь небольшого участка (в худшем случае — одного элемента), что может повлечь глубину рекурсии вплоть до $N-1$ (при N элементах). Это опасная ситуация,

поскольку при больших N происходит переполнение стека, что обычно влечет «падение» программы. К счастью, от возможного переполнения стека довольно легко избавиться — для этого следует представить один из рекурсивных вызовов (на бóльшей части) в виде хвостовой рекурсии и раскрыть его в цикл.

Пример 18.16. Быстрая сортировка с одним вызовом

```
template <class T>
void quick_sort(T * from, T * to) {
  while (to - from > 1) {
    T const key = from[(to - from)/2];
    T * p = key_part2_inplace(from, to, key);
    if (p - from < to - p) {
      quick_sort(from, p);
      from = p;
    } else {
      quick_sort(p , to);
      to = p;
    }
  }
}
```

Конечно, данный прием никак не изменяет количество операций сравнения и обмена, выполняемых сортировкой. В худшем случае имеем отделение по одному элементу в цикле и **квадратичное время** работы алгоритма. Из-за этого было бы справедливо поставить данный алгоритм в предыдущий подраздел, ведь он не гарантирует нам линеарифмическое время работы (хотя оно и таково *в среднем*).

Ситуацию можно исправить, если за линейное время находить медиану или достаточно близкий к ней элемент и использовать его в качестве ключа. На практике в качестве эвристики популярен выбор медианы трех элементов (первый, средний, последний), девяти элементов (по три элемента в каждой трети) или 25 элементов (пять пятерок).

Как и в случае сортировки слиянием, естественной оптимизацией является переход на сортировку вставками при до-

стижении достаточно малого диапазона. Воплотим сказанное в коде:

```
// Медиана трех элементов.
template <class T>
T const & med3(
    T const & a,
    T const & b,
    T const & c) {
  switch (
     (a < b)       |
    ((a < c) << 1) |
    ((b < c) << 2)) {
  case 0: case 7: return b;
  case 3: case 4: return c;
  default:        return a;
  }
}

template <class T>
void quick_sort(T * from, T * to) {
  while (to - from > 30) {
    T const key = med3(
      from[0], from[(to - from)/2], to[-1]);
    T * p = key_part2_inplace(from, to, key);
    if (p - from < to - p) {
      quick_sort(from, p);
      from = p;
    } else {
      quick_sort(p , to);
      to = p;
    }
  }

  insert_sort(from, to);
}
```

В случае, если последовательность содержит много эквивалентных элементов, то обычной быстрой сортировке это лишь вредит. Впрочем, можно разделять элементы не на две части, а на три: меньшие, эквивалентные, бóльшие ключа. Для этого

требуется изменить функцию, выполняющую разделение. Например, так, как показано в примере ниже.

Пример 18.17. Быстрая сортировка, разделение на три части

```
// Разделение элементов на три части по ключу,
//    принадлежащему диапазону.
template <class T>
std::pair<T*, T*> key_part3_inplace(
    T * from, T * to,
    T const & key) {
  assert(from != to);
  assert(find(from, to, key) != to);
  while (*from < key) ++from;
  for (T * p = from; p != to;) {
    if (*p < key)
      std::swap(*from++, *p++);
    else if (key < *p)
      std::swap(*p, *--to);
    else
      ++p;
  }
  return { from, to };
}

template <class T>
void quick3_sort(T * from, T * to) {
  while (to - from > 30) {
    T const key = med3(
      from[0], from[(to - from)/2], to[-1]);
    auto [L, R] = key_part3_inplace(from, to, key);
    if (L - from < to - R) {
      quick3_sort(from, L);
      from = R;
    } else {
      quick3_sort(R , to);
      to = L;
    }
  }
  insert_sort(from, to);
}
```

У Определите, является ли quick3_sort из листинга 18.17 устойчивой сортировкой.

Так как эвристика не гарантирует линеарифмическое время работы (под нее можно *специально* сконструировать плохие входные данные), то был предложен «гибридный» метод на основе объединения быстрой и пирамидальной сортировок, известный как Introsort[150]. Идея метода заключается в отслеживании количества разбиений и переходе к пирамидальной сортировке при достижении определенной границы.

Пример 18.18. Возможный вариант Introsort

```
template <class T>
void intro_sort(T * from, T * to) {
  for (auto cred = to - from;
            cred != 0; // cred = 3*cred/4
            cred = (cred>>1) + (cred>>2)) {
    if (to - from < 30)
      return insert_sort(from, to);
    T const key = med3(
      from[0], from[(to - from)/2], to[-1]);
    auto [L, R] = key_part3_inplace(from, to, key);
    if (L - from < to - R) {
      intro_sort(from, L);
      from = R;
    } else {
      intro_sort(R , to);
      to = L;
    }
  }

  heap3_sort(from, to);
}
```

[150] «Интроспективная сортировка», см.: *Musser D.* Introspective Sorting and Selection Algorithms // Software: Practice and Experience. 1997. № 27(8). P. 983–993.

Данный метод гарантирует линеарифмическое время в худшем случае, а в среднем близок быстрой сортировке. Конечно, Introsort не является устойчивой сортировкой.

18.4.3. Линейные алгоритмы

Выше мы могли видеть пользу от квадратичных алгоритмов, которые, хотя и проигрывают в среднем линеарифмическим сортировкам на больших наборах, способны «брать рекорды» на малых или почти отсортированных наборах, и сортировка вставками используется весьма широко как составная часть более сложных алгоритмов.

Но если есть линейные алгоритмы сортировки, то зачем нужны прочие? Дело в том, что линейные алгоритмы имеют принципиальное отличие от уже рассмотренных алгоритмов: их линейность базируется на использовании расширенного набора операций (возможен даже полный отказ от использования обычных сравнений) или существенной ограниченности множества значений сортируемых элементов.

Можно доказать, что в худшем случае алгоритм сортировки, использующий только сравнения, не может быть асимптотически быстрее линеарифмического. Упрощая, можно сказать, что это следует из того, что каждое сравнение соответствует выбору между двумя возможными перестановками элементов. Каждый такой выбор есть узел дерева решений, включающего все возможные перестановки элементов (ведь сортировка может быть единственной, а исходная последовательность — перемешана любым образом). Всего перестановок N элементов $N!$, и двоичный поиск в худшем случае требует около $\log N!$ операций выбора. Можно показать, что

$$\lim_{N\to\infty} \frac{\log N!}{N \log N} = 1,$$

откуда и возникает линеарифмическое число операций сравнения в худшем случае.

Сортировка подсчетом

Сортировка подсчетом позволяет быстро отсортировать последовательность, элементы которой распадаются на малое заранее известное (или надежно оцениваемое сверху) число классов эквивалентности.

Пример: надо отсортировать последовательность, содержащую целые числа от 0 до 10. Для этого достаточно *посчитать*, сколько раз встречается каждое из них.

Пример 18.19. Элементарная сортировка подсчетом

```
template <int LO = 0, int HI = 10>
void fixed_count_sort(int * from, int * to) {
  // Счетчики.
  size_t counter[HI - LO] {};
  // Подсчет.
  for (int * p = from; p != to; ++p) {
    assert(LO <= *p && *p < HI);
    counter[*p - LO]++;
  }
  // Оформление результата.
  for (int n = 0; n < HI - LO; ++n) {
    int const val = n + LO;
    while (counter[n]--)
      *from++ = val;
  }
}
```

Такой пример может показаться несколько надуманным, однако представим себе простую ситуацию: требуется отсортировать адреса по районам города.

Для этого пронумеруем районы в желаемом порядке (например, алфавитном). После чего создадим массив списков, индекс в котором есть номер района. Теперь достаточно пройти по всем адресам и добавить каждый в список по индексу района, к которому он относится. При необходимости можно затем выполнить сортировку каждого списка по отдельности. После чего выполнить конкатенацию списков.

Алгоритм сортировки подсчетом является базовым для алгоритмов, рассмотренных далее.

Блочная сортировка

Блочная сортировка разбивает пространство значений на *блоки* [*buckets*], «раскладывает» значения по блокам, затем сортирует каждый блок и выполняет их конкатенацию. Сортировка может выполняться рекурсивно, если блок получился большой, или любым эффективным алгоритмом. На практике выбирают блок размера, сопоставимого с кэшем процессора.

Алгоритм оказывается линейным по числу элементов, если размер блока ограничен константой (независимо от алгоритма сортировки самого блока). Поэтому блочная сортировка чувствительна к распределению элементов последовательности: например, если бóльшая их часть сосредоточена около одной точки и попадает в пару блоков, а сотни других блоков остаются почти пустыми, то блочная сортировка вполне может проиграть линеарифмическим алгоритмам, так как все вычисления и раскладывание по блоком окажутся, по сути, пустыми затратами времени.

Если выполняется сортировка массива, то конкатенацию выполнять не требуется, так как блоки размещаются в примыкающих друг к другу диапазонах массива. Сначала нужно вычислить размеры блоков, что выполняется с помощью массива счетчиков, как в примере сортировки подсчетом выше. Затем суммированием счетчиков вычисляются смещения блоков в массиве. Так как требуется разбрасывать элементы в произвольные позиции (каждый следующий элемент может попасть в любой блок), то выделяют буфер того же размера, что и исходный массив, в который копируют элементы сразу в нужное место (кстати, наличие буфера позволяет использовать внутри каждого блока сортировку слиянием), а в конце копируют буфер целиком в исходный массив.

В качестве примера приведем сравнительно простой вариант нерекурсивной блочной сортировки, рассчитанной на рабо-

ту с числами с плавающей запятой (сортировка содержимого блоков выполняется стандартной сортировкой, определенной в заголовочном файле **algorithm**).

Пример 18.20. Блочная сортировка массива чисел с плавающей запятой

```
// Нерекурсивная блочная сортировка.
template <class Float>
void nonrec_bucket_sort(Float * from, Float * to) {
  // Размер блока — 32кб.
  using I = std::ptrdiff_t;
  static const I bucket_size = 32768/sizeof(Float);

  I const items = to - from;
  if (items < 2*bucket_size)
    return std::sort(from, to);

  // Определить минимум и максимум.
  Float minv = *from, maxv = *from;
  for (I i = 1; i < items; ++i) {
    minv = from[i] < minv ? from[i] : minv;
    maxv = maxv < from[i] ? from[i] : maxv;
  }
  if (minv == maxv)
    return;

  // Количество блоков.
  I const buckets = 1 + items/bucket_size;
  // Множитель для вычисления номера блока.
  Float const factor = (buckets - 1)/(maxv - minv);

  // Выделить вспомогательную память:
  //    буфер для копирования значений +
  //    счетчики блоков:
  assert(items < PTRDIFF_MAX / (2*sizeof(Float)));
  I const storage_size = items*sizeof(Float)
                       + buckets*sizeof(I);
  void * const storage = std::malloc(storage_size);
  if (!storage)
    return sort(from, to);
```

```
  Float * const buf = static_cast<Float*>(storage);
  // Размеры блоков (смещения — offsets)
  I * const off = static_cast<I*>(
                    (void*)(buf + items));
  // Обнулить счетчики.
  std::memset(off, 0, buckets*sizeof(I));

  // Вычислить размеры блоков.
  for (I i = 0; i < items; ++i)
    off[(I)(factor * (from[i] - minv))]++;

  // Вычислить начальные значения позиций записи.
  for (I i = 0, s = 0; i < buckets; ++i) {
    I const t = off[i] + s;
    off[i] = s;
    s = t;
  }

  // Раскидать значения по блокам.
  for (I i = 0; i < items; ++i)
    buf[off[(I)(factor * (from[i] - minv))]++]
       = from[i];

  // Отсортировать блоки.
  for (I i = 0, prev = 0; i < buckets; ++i) {
    std::sort(buf + prev, buf + off[i]);
    prev = off[i];
  }

  // Скопировать блоки в исходный массив.
  std::memcpy(from, buf, items*sizeof(Float));
  std::free(storage);
}
```

Поразрядная сортировка

Поразрядная сортировка разбивает каждый элемент на разряды (цифры) и выполняет либо устойчивую сортировку по

каждому разряду от младших (least significant digit, LSD) к старшим, либо сортировку по каждому разряду от старших (most significant digit, MSD) к младшим, независимо сортируя полученные для одного разряда группы значений.

При сортировке неотрицательных чисел или чисел без знака можно просто разбить двоичное представление числа на группы (например, по 11) бит и значение каждой группы трактовать как разряд (номер блока).

Если требуется сортировать числа со знаком, то чтобы получить корректное расположение отрицательных чисел, может потребоваться преобразование значения. Например, в случае представления чисел в дополнительном коде достаточно обратить старший бит числа.

Пример 18.21. LSD поразрядная сортировка массива int32_t

```
// От младших к старшим.
void lsd_radix_sort(int32_t * from, int32_t * to) {
  // 11-битные разряды: 32/11 == 3 разряда.
  static const int
    digit_bits = 11,
    last_digit_bits = 10,
    buckets = 1 << digit_bits,
    last_buckets = 1 << last_digit_bits;
  static const uint32_t
    mask = buckets - 1,
    last_mask = last_buckets - 1;

  using I = std::ptrdiff_t;
  I const items = to - from;
  if (items < buckets)
    return std::sort(from, to);

  // Создать буфер.
  auto const storage =
    std::make_unique<int32_t[]>(items);
  auto const buf = storage.get();

  // Сразу посчитаем по всем разрядам.
```

```
I counters[2*buckets + last_buckets] {};
I * const offs_1 = counters;
I * const offs_2 = offs_1 + buckets;
I * const offs_3 = offs_2 + buckets;

// Фаза 0: заполнить offs_1, offs_2, offs_3.
for (I i = 0; i < items; ++i) {
  uint32_t const
    d0 = from[i],
    d1 =  d0 >> digit_bits,
    d2 = (d0 >> 2*digit_bits) - last_buckets/2;
  offs_1[d0 & mask]++;
  offs_2[d1 & mask]++;
  offs_3[d2 & last_mask]++;
}

for (I i = 0, s1 = 0, s2 = 0; i < buckets; ++i) {
  I const
    t1 = offs_1[i] + s1,
    t2 = offs_2[i] + s2;
  offs_1[i] = s1;
  offs_2[i] = s2;
  s1 = t1;
  s2 = t2;
}

for (I i = 0, s = 0; i < last_buckets; ++i) {
  I const t = offs_3[i] + s;
  offs_3[i] = s;
  s = t;
}

// Фаза 1: упорядочить по младшему разряду.
for (I i = 0; i < items; ++i) {
  uint32_t const item = from[i];
  buf[offs_1[item & mask]++] = item;
}

// Фаза 2: упорядочить по среднему разряду.
for (I i = 0; i < items; ++i) {
```

```
    uint32_t const item = buf[i];
    from[
      offs_2[(item >> digit_bits) & mask]++
        ] = item;
  }

  // Фаза 3: упорядочить по старшему разряду.
  for (I i = 0; i < items; ++i) {
    uint32_t const
      item = from[i],
      d2 = (item >> 2*digit_bits) - last_buckets/2;
    buf[ offs_3[d2 & last_mask]++ ] = item;
  }

  // Скопировать буфер обратно в исходный массив.
  std::memcpy(from, buf, items*sizeof(int32_t));
}
```

При сортировке чисел с плавающей запятой в формате IEEE-754 преобразование существенно сложнее, но тоже возможно. Например, в случае двойной точности:

```
uint64_t radix_sort_transform(double val) {
  auto const raw = bit_cast<uint64_t>(val);
  return raw ^ ((0 - (raw >> 63)) |
                (uint64_t(1) << 63));
}
```

Обратное преобразование:

```
double radix_sort_transform(uint64_t val) {
  auto const raw = val ^ (((val >> 63) - 1) |
                          (uint64_t(1) << 63));
  return bit_cast<double>(raw);
}
```

Здесь функция bit_cast взята из будущего стандарта C++20. В случае отсутствия стандартного определения можно определить ее так, как показано в примере 18.22.

Пример 18.22. Сменить тип, сохранив двоичное представление

```
template <class To, class From> inline
To bit_cast(From const & val) noexcept {
  static_assert(sizeof(From) == sizeof(To));
  static_assert(alignof(From) <= alignof(To));
  // std::is_trivial_v из <type_traits>
  static_assert(std::is_trivial_v<To>);
  To result;
  std::memcpy(&result, &val, sizeof(To));
  return result;
}
```

Использование для той же цели объединения (**union**) или смены типа указателя по стандарту C++ влечет неопределенное поведение (кроме случая, когда указатель приводится к указателю на **char**, **unsigned char** или byte).

Далее применимы те же приемы, что и в случае целых чисел. Тексты программ здесь опустим.

MSD поразрядная сортировка может оказаться еще быстрее, чем LSD сортировка: поскольку массив разбивается по значениям разряда, а затем каждый диапазон обрабатывается независимо, то кэши современных процессоров работают с более высокой эффективностью.

Здесь мы опустим реализации MSD сортировок для чисел, но приведем для того случая, к которому LSD сортировка вовсе неприменима: сортировка си-строк произвольной длины.

Начиная с первого символа строк и до последнего, можно выполнить подсчет символов, упорядочить строки по символам и затем обработать поддиапазон каждого символа рекурсивно, кроме нулевого символа, так как строки, туда попавшие, закончились (поэтому данным методом сортировать си-строки особенно удобно).

«Головная» функция сортировки представлена в примере 18.23.

Пример 18.23. MSD ASCII си-строки

```
void asciz_msd_radix_sort(
    char const ** from, char const ** to) {
  auto const items = to - from;
  if (items < 256)
    return std_sort(from, to);
  // Создать буфер и отсортировать рекурсивно.
  auto const buf =
    std::make_unique<char const*[]>(items);
asciz_msd_radix_sort_inner(from, to, buf.get(), 0);
}
```

Функция std_sort есть обертка стандартной функции сортировки (определена в **algorithm**):

```
void std_sort(char const **from, char const **to) {
  std::sort(from, to,
    [](char const * a, char const * b)
      { return std::strcmp(a, b) < 0; });
}

// Сравнение суффиксов строк a и b, начиная с позиции offset.
inline bool asciz_lt(
    char const * a, char const * b,
    std::ptrdiff_t offset) {
  return std::strcmp(a + offset, b + offset) < 0;
}

// Стандартная сортировка со сдвигом.
void std_sort(
  char const ** from, char const ** to,
  std::ptrdiff_t depth) {
  std::sort(from, to,
    [depth](char const * a, char const * b)
    { return asciz_lt(a, b, depth); });
}
```

Основная работа выполняется рекурсивно вызываемой функцией asciz_msd_radix_sort_inner. Параметр depth есть номер текущего символа, он же — глубина рекурсии. Поскольку неограниченная глубина рекурсии может привести к перепол-

нению стека, по достижении заданной (константой max_depth) глубины происходит переход к стандартной сортировке.

```
void asciz_msd_radix_sort_inner(
    char const ** from, char const ** to,
    char const ** buf,
    std::ptrdiff_t depth) {
  static int const
    max_depth = 24,
    buckets = 128; // в ASCII только 128 символов!
  using I = std::ptrdiff_t;
  I const items = to - from;

  // Если блок мал или достигнута макс. глубина,
  // то отсортировать его стандартной функцией.
  if (items < buckets || depth == max_depth) {
    std_sort(from, to, depth);
    if (depth % 2) // скопировать обратно
      std::memcpy(buf, from, items*sizeof(char**));
    return;
  }
  // Вычислить размеры блоков.
  I offs[buckets] {};
  for (I i = 0; i < items; ++i) {
    unsigned char const bucket = from[i][depth];
    offs[bucket]++;
  }
  // Вычислить стартовые позиции.
  for (I i = 0, s = 0; i < buckets; ++i) {
    I const t = offs[i] + s;
    offs[i] = s;
    s = t;
  }
  // Разбросать по блокам в буфер.
  for (I i = 0; i < items; ++i) {
    auto const item = from[i];
    unsigned char const bucket = item[depth];
    buf[offs[bucket]++] = item;
  }
  // Вернуть закончившиеся строки обратно.
```

```
  if ((depth & 1) == 0)
    std::memcpy(from, buf, offs[0]*sizeof(char**));

  // Обработать рекурсивно непустые строки.
  ++depth;
  for (I i = 1; i < buckets; ++i)
    asciz_msd_radix_sort_inner(
      buf + offs[i-1], buf + offs[i],
      from + offs[i-1], depth);
}
```

Существенный вклад в быстродействие данной сортировки дает отбрасывание совпадающих префиксов строк — в отличие от стандартной сортировки, использующей функцию strcmp. Если применить std::sort к std::string, то может оказаться, что авторы конкретной реализации уже применили подобную оптимизацию и массив std::string будет отсортирован в 2–3 раза быстрее аналогичного массива си-строк.

Определенную проблему представляет количество различных символов на каждом этапе. Если оно существенно меньше количества блоков (128 для ASCII в примере выше), то мы будем терять время на пробегании массива блоков. Нередко бывает так, что вначале блоки заполнены хорошо, но с каждым следующим символом разнообразие падает.

Другая проблема заключается в использовании кодировок с большим количеством символов. Основная такая кодировка — Unicode. Слишком большое количество блоков, естественно, убьет производительность, даже если нам хватит памяти в стеке для размещения массива счетчиков.

Интересной альтернативой поразрядной сортировке на основе подсчета символов выступает трехсекционная быстрая сортировка на основе сравнения символов. Она также позволяет отсекать общий префикс строк — все строки, попавшие в среднюю секцию, имеют одинаковый символ в текущей позиции. Недостатки классической быстрой сортировки, естественно, никуда не пропадают, зато вышеперечисленные проблемы для нее не существуют. Приведем код.

Процедура разделения диапазона на три секции получается элементарной модификацией кода, приведенного в предыдущем разделе: в качестве ключа передаем символ и передаем дополнительный параметр — «глубину» символа.

Пример 18.24. Процедура разделения си-строк

```
std::pair<char const**, char const**>
asciz_key_part3_inplace(
    char const ** from, char const ** to,
    char key, std::ptrdiff_t depth) {
  while (from != to && (*from)[depth] < key)
    ++from;
  for (auto p = from; p != to;) {
    char const pch = (*p)[depth];
    if (pch < key)
      std::swap(*from++, *p++);
    else if (key < pch)
      std::swap(*p, *--to);
    else
      ++p;
  }
  return { from, to };
}
```

Теперь представим код самой сортировки.

Пример 18.25. Быстрая сортировка си-строк

```
// Быстрая поразрядная сортировка
// с разделением на три поддиапазона.
void asciz_quick3_sort(
    char const ** from, char const ** to,
    std::ptrdiff_t depth = 0) {
  while (to - from > 20) {
    auto const key = med3(
      from[0][depth],
      from[(to - from)/2][depth],
      to[-1][depth]);

    auto [l, r] = asciz_key_part3_inplace(
        from, to, key, depth);
```

```
    // Теперь у нас три участка:
    // [from, l) [l, r) [r, to).
    // Найдем наибольший из них.
    // Первый?
    if (r - l <= l - from && to - r <= l - from) {
      // Второй и третий рекурсивно.
      asciz_quick3_sort(r, to, depth);
      if ((*l)[depth] != '\0')
        asciz_quick3_sort(l, r, depth + 1);
      // Перейдем к первому.
      to = l;
    }
    // Второй?
    else if (l - from <= r - l && to - r <= r - l) {
      // Первый и третий рекурсивно.
      asciz_quick3_sort(from, l, depth);
      asciz_quick3_sort(r, to, depth);
      // Перейдем ко второму.
      if ((*l)[depth] == '\0')
        return;
      from = l;
      to = r;
      ++depth;
    }
    // Третий.
    else {
      // Второй и первый рекурсивно.
      asciz_quick3_sort(from, l, depth);
      if (l != r && (*l)[depth] != '\0')
        asciz_quick3_sort(l, r, depth + 1);
      // Перейдем к третьему.
      from = r;
    }
  }

  asciz_insert_sort(from, to, depth);
}
```

У Реализуйте функцию asciz_insert_sort, выполняющую сортировку вставками массива си-строк, пропуская при сравнении префикс длины depth.

18.5. Исключения

Исключения [*exceptions*] в C++ представляют собой механизм нелокальной передачи управления, предназначенный для обработки ошибок, которая не может быть выполнена в месте возникновения ошибки. Не рекомендуется использовать исключения для других целей, поскольку компиляторы обычно исходят из предположения, что исключительные ситуации действительно *исключительны*, т. е. возникают редко.

Данный раздел не содержит полное описание стандартного механизма исключений C++ и связанных с ним элементов стандартной библиотеки C++. Его цель — предоставить начальные сведения об исключениях C++.

Исключения позволяют покинуть функцию, не возвращая значения, их поддержка влияет на принцип работы со стеком вызовов. Если функция использует исключения, она несовместима с ABI языка C (который исключения не поддерживает). Соответственно, функции, написанные на C или иные C ABI-совместимые функции, не могут использовать исключения.

Итак, исключение — это значение, которое можно *бросить* [*throw*] в одном месте кода и *поймать* [*catch*] в другом месте, называемом *обработчиком* [*handler*] исключения. Таким образом, управление в случае бросания передается обработчику, а назад уже не возвращается. Бросание исключения оформляется в виде оператора **throw** (может быть частью выражения):

throw выражение

В C++ можно бросить значение любого типа.

Поимка исключения оформляется в виде инструкции, состоящей из **try**-блока, при исполнении которого могут быть брошены перехватываемые исключения, и последовательности обработчиков, каждый из которых начинается с ключевого

слова **catch**. Последовательность обработчиков должна содержать, по крайней мере, один обработчик:

```
try
  блок, бросающий исключения
catch (объявление-1)
  блок обработчика-1
catch (объявление-2)
  блок обработчика-2
// ...
catch (объявление-n)
  блок обработчика-n
```

Принцип действия данной инструкции напоминает принцип действия **switch-case**: в случае бросания исключения выполняется первый (сверху вниз) обработчик, объявление которого можно связать с брошенным значением, так, чтобы тип совпадал (без учета ссылок и **const**). Например, следующий код должен вывести «i1»:

```
try { throw 1; /* int */ }
catch (float f) { cout << "f" << f; }
catch (int i) { cout << "i" << i; }
```

При обработке исключений происходит освобождение кадров стека вызова (*разворачивание стека* [*stack unwinding*]) до тех пор, пока не будет найден подходящий обработчик или стек вызовов не будет опустошен. В последнем случае происходит вызов стандартной функции terminate, завершающей работу программы.

Освобождение кадров стека в процессе разворачивания влечет вызов деструкторов всех полностью созданных объектов и является одной из особенностей языка C++. По этой причине, например, память, на которую указывают unique_ptr, будет освобождена. Простой пример:

```
auto a = new char[n];
auto b = new char[m];
// ...
delete[] b;
delete[] a;
```

В случае бросания исключения вторым **new** имеем утечку памяти, так как **delete[]** a не будет выполнен. При использовании unique_ptr память будет освобождена автоматически, в том числе и при бросании исключения:

```
auto a = make_unique<char[]>(n);
auto b = make_unique<char[]>(m);
// delete не нужны
```

Конструкции **try-catch** могут быть вложенными в любых комбинациях. Если для брошенного исключения не найдется обработчик внутри вложенной инструкции **try-catch**, то она будет покинута и будет произведена попытка найти подходящий обработчик в содержащей ее инструкции **try-catch** и т. д. Внутри **catch** также можно бросать исключения. Более того, можно даже перебросить исходное пойманное исключение далее вызовом **throw** без аргумента:

```
throw; // только внутри блока catch
```

Условный пример:

```
int f(int a) {
  switch (a % 3) {
  case 0: throw a;
  case 1: throw "hello!";
  default: return a;
  }
}

int g(int a) {
  try { return f(a) + 1; }
  catch (int) { return 0; }
}

int main() {
  int n = 0;
  cin >> n;
  try { cout << g(n); }
  catch (char const * msg) { cout << msg; }
}
```

Обработчик исключения любого типа (без доступа к брошенному значению) определяется с помощью особого синтаксиса:

```
catch (...) { cout << "an exception occured"; }
```

Естественно, такой обработчик всегда должен стоять в конце последовательности обработчиков (аналогично секции **default** внутри блока **switch**).

По умолчанию считается, что вызов любой функции может повлечь бросание исключения. Это, естественно, касается и переопределенных операторов. Например, в выражении ниже имеем до четырех потенциальных источников исключений:

```
out << f() + g();
```

Операции со встроенными (а не библиотечными) типами не бросают исключений, даже если действие ошибочное. Например, целочисленное деление на нуль влечет неопределенное поведение, его нельзя перехватить с помощью **try-catch**. Это касается всех ситуаций неопределенного поведения — бросание исключения есть определенное поведение!

Функции можно объявить явно как не бросающие исключения с помощью ключевого слова **noexcept**, которое ставится после списка параметров:

```
void run() noexcept;
```

При этом функция может внутри себя организовать перехват исключений — главное, чтобы они не могли «вырваться наружу». В противном случае исключение не будет передано дальше, вместо этого будет сразу вызвана функция terminate (это тоже определенное поведение).

Всякий деструктор всегда считается не бросающим исключения (неявно объявлен **noexcept**).

Конструктор может бросать исключения, но в этом случае объект не считается полностью созданным, и деструкторы будут вызваны только для тех полей, для которых были успешно выполнены конструкторы. Об остальном следует позаботиться программисту.

Объявление конструкторов и некоторых других функций **noexcept** может позволить компилятору и стандартной библиотеке задействовать определенные оптимизации, но это следует делать, учитывая, что **noexcept** является не деталью реализации, а частью интерфейса: это — гарантия, которую дает программист.

Поскольку исключения могут возникать в конструкторах полей, то была введена особая форма **try-catch**, позволяющая перехватывать эти исключения размещением списка инициализации под **try**:

```
class A {
  B b; C c;
  // Список инициализации под try:
  A() try : b(), c() {
    // Все тело под try.
  }
  // Произвольная последовательность обработчиков.
  catch (...) {
    // Обработчик исключений.
  }
// ...
```

Конструкцию **try-catch** также можно использовать вместо тела обычной функции:

```
void foo() // {
try {
  // Тело.
} catch (...) {
  // Обработчик.
}
// }
```

В книге были рассмотрены некоторые компоненты, бросающие исключения:

- оператор **new** в случае невозможности выделения памяти и при отсутствии параметра nothrow бросает исключение типа bad_alloc (определен в заголовочном файле **new**);

- функции распознавания записей числовых констант stoi, stol, stof и т. д. бросают исключения следующих типов (определены в **stdexcept**):
  - invalid_argument если не удалось распознать запись числа;
  - out_of_range если запись числа не может быть представлена возвращаемым функцией типом, поскольку не попадает в его множество значений;
- функции at и substr класса string_view в случае, если переданный индекс позиции (начала подстроки в случае substr) выходит за пределы диапазона, бросают исключение out_of_range.

Список рекомендуемой литературы и источников

C++ reference : [сайт]. — URL: `https://cppreference.com` (дата обращения: 25.07.2020).

Standard C++ : [сайт]. — URL: `https://isocpp.org` (дата обращения: 25.07.2020).

Ахо А. Компиляторы: принципы, технологии и инструментарий / А. Ахо, М. Лам, Р. Сети, Дж. Ульман. — 2-е изд. — Москва : Вильямс, 2008. — 1184 с. — ISBN 978-5-8459-1349-4.

Брукс Ф. Мифический человеко-месяц, или Как создаются программные системы / Ф. Брукс. — Санкт-Петербург : Символ-Плюс, 2007. — 304 с. — ISBN 978-5-93286-005-2.

Готтшлинг П. C++ для инженерных и научных расчетов / П. Готтшлинг. — Санкт-Петербург : ООО «Диалектика», 2020. — 512 с. — ISBN 978-5-907203-30-3.

Игошин В. И. Теория алгоритмов : учеб. пособие / В. И. Игошин. — Москва : ИНФРА-М, 2016. — 318 с. — ISBN 978-5-16-005205-2.

Керниган Б. Язык программирования C / Б. Керниган, Д. Ритчи. — 2-е изд. — Москва : Вильямс, 2019. — 288 с. — ISBN 978-5-8459-1975-5.

Петцольд Ч. Код : тайный язык информатики / Ч. Петцольд. — Москва : Русская Редакция, 2004. — 512 с. — ISBN 5-7502-0159-7.

Роуз Д. От математики к обобщенному программированию / Д. Роуз, А. А. Степанов. — Москва : ДМК Пресс, 2015. — 264 с. — ISBN 978-5-97060-289-8.

Спрингер В. Гид по Computer Science для каждого программиста / В. Спрингер. — Санкт-Петербург : Питер, 2020. — 192 с. — ISBN 978-5-4461-1674-4.

Степанов А. А. Начала программирования / А. А. Степанов, П. Мак-Джонс. — Москва : Вильямс, 2011. — 272 с. — ISBN 978-5-8459-1708-9.

Страуструп Б. Программирование : принципы и практика использования C++ / Б. Страуструп. — 2-е изд. — Москва : Вильямс, 2016. — 1328 с. — ISBN 978-5-8459-1949-6.

Стюарт Т. Теория вычислений для программистов / Т. Стюарт. — Москва : ДМК Пресс, 2014. — 384 с. — ISBN 978-5-94074-979-0.

Уильямс Э. C++. Практика многопоточного программирования / Э. Уильямс. — Санкт-Петербург : Питер, 2020. — 640 с. — ISBN 978-5-4461-0831-2.

Учебное издание

Кувшинов Дмитрий Рустамович
Осипов Сергей Иванович

ОСНОВЫ ПРОГРАММИРОВАНИЯ
Язык C++

Учебное пособие

Заведующий редакцией	М. А. Овечкина
Редактор	С. Г. Галинова
Корректор	С. Г. Галинова
Компьютерная верстка	Д. Р. Кувшинов

Подписано в печать 01.07.2021 г. Формат $60 \times 84^{1}/_{16}$.
Бумага офсетная. Цифровая печать. Усл. печ. л. 28,61.
Уч.-изд. л. 21,0. Тираж 30 экз. Заказ 111.

Издательство Уральского университета
Редакционно-издательский отдел ИПЦ УрФУ
620083, Екатеринбург, ул. Тургенева, 4
Тел.: +7 (343) 389-94-79, 350-43-28
E-mail: rio.marina.ovechkina@mail.ru

Отпечатано в Издательско-полиграфическом центре УрФУ
620083, Екатеринбург, ул. Тургенева, 4.
Тел.: +7 (343) 358-93-06, 350-58-20, 350-90-13
Факс +7 (343) 358-93-06
http://print.urfu.ru

Для заметок

Для заметок


УНИВЕРСИТЕТСКОГО КНИГОИЗДАНИЯ
2020


9 785799 632571