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

Д. Р. КУВШИНОВ

КОМПЬЮТЕРНЫЕ НАУКИ
Основы программирования

Учебное пособие

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
УРАЛЬСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ
ИМЕНИ ПЕРВОГО ПРЕЗИДЕНТА РОССИИ Б. Н. ЕЛЬЦИНА

Д. Р. Кувшинов

КОМПЬЮТЕРНЫЕ НАУКИ

Основы программирования

Рекомендовано методическим советом УрФУ
в качестве учебного пособия для студентов, обучающихся
по программе бакалавриата по направлению подготовки
010800 «Механика и математическое моделирование»

Екатеринбург
Издательство Уральского университета
2015

УДК 004.42(075.8)
К885

Р е ц е н з е н т ы :
кафедра информатики и математики
Уральского института управления — филиала РАНХиГС
(заведующий кафедрой — доктор физико-математических наук
профессор С. Ю. Ш а ш к и н);
А. В. Г о р ш к о в, кандидат физико-математических наук, доцент
(Институт машиноведения УрО РАН)

Кувшинов, Д. Р.

К885 Компьютерные науки : Основы программирования : [учеб. пособие] / Д. Р. Кувшинов ; М-во образования и науки Рос. Федерации, Урал. федер. ун-т. — Екатеринбург : Изд-во Урал. ун-та, 2015. — 104 с.

ISBN 978-5-7996-1411-9

Издание содержит набор теоретических сведений, упражнений и пояснений, предназначенных для формирования и закрепления базовых навыков программирования на языке C++. Отдельное внимание уделено работе с числами с плавающей запятой. Пособие предназначено для студентов, начинающих изучать программирование.

УДК 004.42(075.8)

ISBN 978-5-7996-1411-9

Оглавление

Предисловие

Язык программирования C++ на данный момент является одним из основных инфраструктурных языков программирования и используется в системном программировании, высокопроизводительных вычислениях, при создании промежуточного программного обеспечения (библиотек, служащих основой для прикладного программного обеспечения) и интерактивных приложений, требующих высокого быстродействия (в частности, видеоигр).

Данное пособие предназначено для получения и закрепления базовых навыков структурного программирования на языке C++ в рамках начального курса программирования. Следует признать, что язык C++ сравнительно тяжел при первоначальном освоении, однако имеет не только высокую практическую, но и высокую дидактическую ценность. Овладевшие им студенты в случае необходимости довольно легко осваивают новые языки и технологии программирования, многие из которых исторически связаны с C и C++.

Материал пособия разбит на пять разделов и приложение. Первый раздел «Начало работы» предназначен для ознакомления студентов с созданием простейших программ и стандартными средствами текстового ввода-вывода. Следующие разделы «Арифметика и логика», «Функции», «Массивы и циклы» и «Структуры данных и файлы» последовательно наращивают синтаксическую и семантическую нагрузку упражнений для постепенного освоения основных элементов парадигмы струк-

турного программирования на основе C++. Приложение содержит два раздела. Раздел «Числа с плавающей запятой» посвящен стандарту IEEE-754, реализуемому в современном аппаратном и программном обеспечении и наиболее широко применяемому при решении практических задач с использованием численных методов и имитационного моделирования. Раздел «Справочный материал по cmath» содержит таблицу стандартных математических функций из заголовочного файла cmath. Для ряда терминов указаны эквиваленты на английском языке (в скобках курсивом).

В каждом разделе приведен соответствующий теоретический материал и набор упражнений. Упражнения отмечены значком [У]. Предполагается, что упражнения из первого раздела будут выполняться студентами в компьютерном классе в самом начале первого семестра обучения с целью ознакомления со средой разработки и простейшими элементами синтаксиса C++. Упражнения из пятого раздела предполагают использование классов и инкапсуляцию внутреннего состояния объектов. Упражнения 2.1–2.12, 2.23, 3.21, 3.22, 3.34, 4.15, 4.16 могут быть использованы на занятиях вне компьютерного класса. Бóльшая же часть упражнений предназначена для использования в составе лабораторных работ. Предлагается разбить их по лабораторным следующим образом.

- Семестр 1: 1) 3.1–3.20; 2) 3.23; 3) 3.24–3.33; 4) 2.13–2.22; 5) 4.1–4.14; 6) 4.17–4.26; 7) 4.28–4.35; 8) 4.36–4.42; 9) 4.43; 10) 4.44–4.50.

- Семестр 2: 1) 5.1, 5.2; 2) 5.3–5.6; 3) 5.7, 5.8; 4) 5.9–5.11; 5) 5.12.

Подробные сведения о стандартных структурах данных и алгоритмах, о программировании на языке C++ и использовании Стандартной библиотеки C++ можно почерпнуть из литературы, список которой приведен в конце пособия.

Глава 1

Начало работы

1.1. Создание проекта

Данное пособие предполагает использование интегрированной среды разработки Microsoft Visual Studio 2012[1]. Чтобы начать программировать в этой среде, требуется создать проект разрабатываемого приложения (программы). Ниже приведена последовательность действий, приводящих к созданию проекта Visual C++ с одним файлом исходного кода C++.

1. Вызвать мастер создания нового проекта: Файл → Создать → Проект...
2. Выбрать тип решения (в случае решения задач из пособия — Visual C++ Консольное приложение Win32).
3. В том же окне ввести название проекта (в примере — HelloWorld) и нажать кнопку OK.
4. В появившемся окне выбрать Параметры приложения.
5. Установить флажок Пустой проект, после чего нажать кнопку Готово.

[1]Допускается использование других компиляторов и сред разработки.

6. В данный момент наш проект пуст. Чтобы вводить C++-код, необходимо добавить хотя бы один .cpp-файл: Проект → Добавить новый элемент...

7. В появившемся окне выбрать Файл C++ (.cpp), в поле ввода внизу ввести название файла (например, «HelloWorld») и нажать кнопку Добавить.

8. Теперь можно набирать код.

Обратите внимание, что по умолчанию проект размещается в отдельной папке, находящейся по адресу Мои документы\Visual Studio 2012\Projects\, название которой совпадает с названием проекта. Файлы с исходным кодом C++ являются обычными текстовыми файлами, при необходимости их можно править даже простейшим текстовым редактором: например, встроенным в ОС Windows Блокнотом. Файлы с исходным кодом размещаются во вложенных папках: например, наш HelloWorld.cpp находится по адресу Мои документы \ Visual Studio 2012 \ Projects \ HelloWorld \ HelloWorld \ HelloWorld.cpp.

1.2. Структура проекта

В среде Visual C++ проект представляет собой иерархическую структуру, которая отображается в окне Обозреватель решений. На вершине этой структуры находится **решение** (*solution*), описание которого хранится в .sln-файле. Решение может объединять несколько связанных проектов, каждый из которых по умолчанию располагается в отдельной папке внутри папки решения. Решение создается автоматически вместе с новым проектом.

Проект, в свою очередь, состоит из файлов и позволяет задать конфигурацию сборки приложения (в нашем случае — .exe-файла). Описание проекта хранится в файлах с расширениями .vcxproj и .vcxproj.filters.

Чтобы настроить конфигурацию решения или проекта, можно щелкнуть правой кнопкой мыши по названию решения или проекта в окне Обозреватель решений и выбрать пункт меню Свойства.

Перед тем как запустить только что написанную C++-программу, ее нужно **собрать**. Процесс сборки проекта состоит из двух основных этапов:

- **компиляция** — перевод всех файлов с исходным кодом, включенных в проект, в машинный код, понятный центральному процессору;
- **компоновка** — объединение результатов в исполняемый файл (.exe в ОС Windows), представляющий собой целостное приложение, которое уже можно запустить на исполнение.

Начать процесс сборки можно различными способами:

- нажав клавишу F7 либо выбрав пункт меню Построение → Построить решение (пересобрать только те части решения, которые изменились с момента предыдущей сборки) или пункт меню Построение → Перестроить решение (выполняет полную пересборку, целиком удаляя результаты предыдущей сборки);
- нажав клавишу F5 или кнопку в виде зеленого треугольника на панели инструментов, что приведет к запуску полученного .exe-файла в режиме отладки, если сборка прошла успешно;
- нажав Ctrl+F5 (см. далее).

Справа от «зеленого треугольника» находится список выбора **конфигурации сборки**. По умолчанию выбрана конфигурация Debug (отладка). При финальной сборке программных продуктов обычно используют конфигурацию Release (выпуск). Эти конфигурации различаются применяемыми компилятором

оптимизациями и наличием отладочной информации. В случае необходимости можно добавить дополнительные конфигурации.

1.3. Простейшая программа

Стандарт C++ строго регламентирует, какой текст может считаться корректной C++-программой, а какой — нет. Например, мы могли бы попробовать что-нибудь посчитать. Наберем в окне редактора `2+2=?` и нажмем F7.

Увы, компилятор не понял, что означает текст «2+2=?», поскольку он не является корректной C++-программой. Последняя строка вывода сообщает нам количество успешно собранных проектов текущего решения (после слова «успешно»). Количество проектов, сборка которых не удалась, указано после фразы «с ошибками» (в нашем случае один проект). Наконец, если какие-то проекты решения не пересобирались (например, в них не было внесено изменений со времени предыдущей сборки), то их количество будет указано после слова «пропущено».

В списке сообщений компилятора мы можем видеть **ошибки** (*errors*) и **предупреждения** (*warnings*). Сборка не будет выполнена успешно, пока есть хотя бы одна ошибка. Предупреждения не препятствуют успешной сборке, но на них следует обращать внимание, так как они могут указывать на семантические ошибки, допущенные программистом. Компилятор предупреждает, что код выглядит подозрительно, но если вы уверены в том, что все правильно, то можете оставить его без изменений.

Сообщение об ошибке `синтаксическая ошибка: константа` возникло из-за того, что корректная C++-программа не может начинаться с константы (в нашем случае — с 2). Двойной щелчок по сообщению об ошибке в окне Вывод позволяет перейти к месту ошибки в тексте программы.

К сожалению, нередко компилятор оказывается неспособным точно определить место синтаксической ошибки (напри-

мер, пропущенной скобки или точки с запятой), поэтому если в точке, указанной компилятором, ошибку найти не удается, то следует проанализировать код, находящийся неподалеку.

Текст программы может состоять из следующих элементов:

- комментарии, игнорируемые компилятором;
- директивы препроцессора, выполняемые до компиляции;
- директивы компилятора;
- объявления;
- определения.

Кроме того, программа, которая компилируется в исполняемый файл, должна содержать **точку входа** (*entry point*) — правильно оформленный фрагмент кода, с которого начинается выполнение при запуске приложения. В случае стандартного C++ точка входа является функцией с именем `main`, минимально возможный вариант которой выглядит следующим образом.

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

Если вы наберете этот код вместо ранее введенного 2+2=? и нажмете F5, то проект будет успешно собран, а полученный в результате сборки .exe-файл — запущен. Так как мы создали проект консольного приложения, то появится окно консоли, с помощью которого можно вводить и выводить текст. Оно сразу закроется, потому что наша программа ничего не делает, завершаясь немедленно после запуска.

Полученный в результате сборки .exe-файл можно запустить и напрямую вне **отладчика** (*debugger*), который запускается по F5 и позволяет отслеживать процесс исполнения программы. Откомпилированные .exe-файлы располагаются внутри папки проекта в папке, названной по конфигурации сборки. В нашем случае, это Мои документы \ Visual Studio 2012 \ Projects \ HelloWorld \ Debug \ HelloWorld.exe.

Строчка в примере есть не что иное, как **определение** (*definition*) функции с именем `main`, не принимающей параметров и возвращающей целое число. Определение задает некоторую именованную сущность, например функцию или переменную. Благодаря наличию имени мы можем сослаться на эту сущность в коде, размещенном ниже определения в той же области видимости.

В то время как определение дает исчерпывающее описание некоторой именованной сущности, **объявление** (*declaration*) лишь вводит имя, сообщая компилятору, что где-то в другом месте кода дано полное определение. В основном объявления используются для связи разных частей программы друг с другом и обычно помещаются в заголовочные файлы.

Некоторое имя может соответствовать не более чем одному определению («правило одного определения»), в то время как идентичных объявлений может быть много.

Комментарии (*comments*) предназначены для размещения в коде заметок в виде обычного текста, содержащих пояснения для людей, читающих программу. Кому-то это может показаться странным, однако программы следует писать в первую очередь для людей, и лишь во вторую очередь — для компьютеров. Если программа написана неряшливо, путано, в ней отсутствуют пояснения и не видна логика, то ее очень трудно отлаживать, развивать и сопровождать, в ней наверняка множество ошибок.

1.4. Текстовый ввод-вывод

Собственно язык C++ как таковой не содержит никаких встроенных средств ввода-вывода, работы с файлами, периферийными устройствами и операционной системой. Все эти средства должны быть взяты из существующих библиотек.

Стандарт C++ определяет минимум функционала, который должен входить в комплект поставки C++-компилятора. Этот минимум называется **Стандартной библиотекой C++**. Что-

бы воспользоваться некоторой частью Стандартной библиотеки, требуется подключить эту часть к .cpp-файлу, в котором предполагается ее использовать. Подключение осуществляется включением необходимых объявлений и определений в текст программы.

Объявления и определения Стандартной библиотеки разбиты на группы, помещенные в отдельные файлы — **стандартные заголовочные файлы**. Например, стандартные средства консольного текстового ввода-вывода размещены в заголовочном файле iostream. Чтобы не копировать объявления как текст и не загромождать ими свою программу, можно сообщить компилятору, что содержимое заголовочного файла требуется вставить в то или иное место вашей программы. Для этого служит директива препроцессора `include`:

```
#include <iostream>
```

Можно считать, что при компиляции все подобные строчки заменяются текстом указанного файла. Слово **препроцессор** («предобработчик») используется потому, что все строчки, начинающиеся с символа # (и называемые **директивами препроцессора**) обрабатываются до начала компиляции основного текста программы. Раньше эта обработка выполнялась отдельной, сравнительно простой программой, которая и называлась «препроцессором». Обработанные препроцессором файлы транслировались в машинный код программой-компилятором.

В примере 1.1 показано, как, используя iostream, вывести в окно консоли строчку текста. Заодно показан синтаксис оформления комментариев в C++.

Пример 1.1. 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!";
}
```

Строчка, выводящая текст приветствия на экран,

```
std::cout << "Hello, world!";
```

состоит из следующих частей:

- имя (глобальной переменной) `std::cout`, привязанное к стандартному (системному) текстовому потоку вывода[2];
- операция `<<`, в данном контексте означающая «вывести то, что справа, в поток, указанный слева»;
- то, что требуется вывести в поток;
- точка с запятой, являющаяся признаком конца инструкции в C++.

Собственно

```
std::cout << "Hello, world!"
```

без точки с запятой является **выражением**, то есть конструкцией, состоящей из значений (имен или непосредственно заданных констант — **литералов**) и связующих их операций.

Например, `2 * a + b` является выражением: `a` и `b` — имена переменных, `2` — литерал, а `*` и `+` — операции. Если `a` и `b` хранят числа, то это выражение по смыслу отражает алгебраическую запись $2a + b$ (умножить `a` на два и добавить `b`).

[2]Этот поток называется *стандартным выводом* (stdout от англ. *standard output*), в C++ `cout` является сокращением от *console output*.

Выражение транслируется в код, вычисляющий значение этого выражения, используя значения переменных, актуальные на тот момент, когда процессор дойдет до исполнения этого кода. Таким образом, выражение имеет значение (в общем случае неизвестное до момента вычисления) и тип (известный на момент компиляции), который позволяет компилятору определить множество возможных значений выражения и способ их представления в памяти компьютера.

Добавление точки с запятой превращает выражение в самостоятельную **инструкцию**, выполнение которой приводит к вычислению значения выражения и выполнению всех связанных с этим **побочных эффектов**. После этого вычисленное значение отбрасывается, смысл же инструкции состоит как раз в «побочных» эффектах. Побочными они являются с точки зрения задачи вычисления значения выражения. Например, вывод строчки `"Hello, world!"` в текстовый поток при выполнении инструкции

```
std::cout << "Hello, world!";
```

является побочным эффектом вычисления выражения до точки с запятой. Значением же этого выражения будет сам поток `std::cout`, что позволяет записать вывод нескольких строк в поток одним выражением. Например,

```
std::cout << "2 * 2 = " << 4;
```

будет понято компилятором как

```
(std::cout << "2 * 2 = ") << 4;
```

Побочным эффектом при вычислении выражения в скобках является вывод строчки `"2 * 2 = "`, после чего написанное становится эквивалентным

```
(std::cout) << 4;
```

и выводит в поток `4`. В итоге получаем вывод `2 * 2 = 4`.

При запуске HelloWorld.exe под ОС Windows двойным щелчком мыши или через отладчик вряд ли удастся успеть прочитать то, что программа вывела в консоль. Для того, чтобы окно

не закрывалось, можно вставить в конце функции `main` операцию чтения символа с клавиатуры.

Пример 1.2. HelloWorld, ожидание ввода символа

```
// используем консольный ввод-вывод
#include <iostream>
// отсюда начинается выполнение программы
int main() {
  // вывести строчку Hello, world!
  std::cout << "Hello, world!";
  // ждем, пока пользователь не введет
  // какой-нибудь символ или не нажмет Enter
  std::cin.get();
}
```

Все определения из Стандартной библиотеки C++ помещены в пространство имен `std`. Конструкция вида `std::`*имя* используется для того, чтобы получать доступ к именам, определенным в пространстве имен `std`.

Чтобы каждый раз не повторять `std::`, можно в своей функции или непосредственно в начале `.cpp`-файла (после всех директив `include`) указать компилятору, что мы используем определения из пространства имен `std` «по умолчанию». Так:

```
#include <iostream>
int main() {
  // после этой строчки
  using namespace std;
  // std:: в этой функции уже можно не писать
  cout << "Hello, world!";
  cin.get();
}
```

Или так:

```
#include <iostream>
// после этой строчки
using namespace std;
```

```
// std:: в этом файле уже можно не писать
int main() {
  cout << "Hello, world!";
  cin.get();
}
```

Добавим в наш пример интерактивности. Будем спрашивать имя пользователя и приветствовать его по имени. Имя — это последовательность символов, иными словами — «текст» или «строка». Тип переменных, значением которых является некоторый текст, называется **строковым типом**. В Стандартной библиотеке C++ есть средства для работы со строками, размещенные в заголовочном файле string.

Пример 1.3. Ввод строки (I)

```
#include <iostream>
// подключить строки C++
#include <string>
using namespace std;
int main() {
  cout << "What is your name? ";
  // имя запишем в строковую переменную name
  string name;
  cin >> name;
  // поприветствуем обладателя имени name
  cout << "Hello, " << name << "!";
  cin.get();
}
```

Здесь строчка

```
string name;
```

является определением **локальной** (видимой только внутри функции, где она определена, и существующей только во время работы этой функции) переменной. Эта переменная служит местом хранения нужного нам текста (имени пользователя) и

дает ему имя, благодаря наличию которого к этому тексту (значению переменной) можно обращаться в программе.

Инструкция `cin >> name;` отвечает за считывание текста из потока ввода. Операция `>>` здесь означает «прочитать из потока, указанного слева, значение и положить его (если возможно) в переменную, имя которой указано справа».

Первая же попытка запуска примера 1.3 выявит неожиданную проблему: кажется, строчка

```
cin.get();
```

перестала действовать. Теперь окно консоли опять закрывается сразу после вывода текста. Причина такого поведения в том, что при считывании строкового значения из текстового потока ввода[3] с помощью операции `>>` пропускаются начальные **пробельные символы** (к которым относятся пробел, символ перевода каретки, табуляция), затем извлекаются все символы до первого пробельного.

Когда пользователь вводит имя и нажимает Enter, в поток ввода отправляются символы введенного имени и символ перевода каретки, являющийся пробельным. Операция `>>` считывает в переменную `name` символы имени, но оставляет символ перевода каретки в потоке. Когда дело доходит до `cin.get()`, происходит считывание одного символа из потока `cin`. Если бы он был пуст, то выполнение программы остановилось бы, ожидая ввода пользователя (как было в предыдущих примерах). Однако теперь он не пуст — в нем есть символ перевода каретки, который и будет успешно считан, поэтому выполнение программы продолжится.

Можно добавить еще одну строчку `cin.get();` после считывания имени, чтобы «превентивно» убрать ожидаемый нами символ перевода каретки.

[3] В данном случае это стандартный поток ввода stdin, который в C++ доступен через переменную `std::cin` — *console input*.

Пример 1.4. Ввод строки (II)

```
#include <iostream>
// подключить строки C++
#include <string>
using namespace std;
int main() {
  cout << "What is your name? ";
  // имя запишем в строковую переменную name
  string name;
  cin >> name;
  // уберем лишний символ перевода каретки
  cin.get();
  // поприветствуем обладателя имени name
  cout << "Hello, " << name << "!";
  cin.get();
}
```

Впрочем, при попытке ввести имя, состоящее из двух и более слов, разделенных пробелами, считано и выведено в приветствии будет только первое, а дополнительный `cin.get()` не предотвратит немедленного закрытия окна после вывода приветствия. Причина та же: операция `>>` читает до пробела, пробел мы забираем вызовом `cin.get()`, в переменной `name` остается первое введенное слово, а остальные слова остаются в потоке `cin`. Завершающий вызов `cin.get()` попросту извлекает первую букву слова, введенного вторым, и не приводит к блокировке.

У (1.1) Изменить пример 1.4 так, чтобы компьютер запрашивал (отдельно) фамилию и имя пользователя и выводил приветствие в форме «Hello, *name surname*!». Кроме переменной `name` требуется завести переменную `surname`.

Вместо того чтобы читать до первого пробельного символа, можно читать все до заданного символа (блокируя до нажатия Enter, если поток `cin` станет пуст, а нужного символа так и не встретится). Такой символ называется **разделителем**

(*delimiter*). Для этого заменим `>>` вызовом стандартной функции `getline`, читающей из потока все символы до заданного символа-разделителя (по умолчанию в качестве разделителя используется перевод каретки). Функция `getline` убирает из потока ввода и сам символ-разделитель, поэтому код в примере 1.5 будет ожидать ввода символа.

Пример 1.5. Ввод строки (III)

```
#include <iostream>
#include <string>
using namespace std;
int main() {
  cout << "What is your name? ";
  string name;
  // считать строчку: откуда (cin), куда (name)
  getline(cin, name);
  // поприветствуем обладателя имени name
  cout << "Hello, " << name << "!";
  cin.get();
}
```

При запуске из среды Visual Studio необходимости добавлять `cin.get()` или что-то аналогичное в код для задержки экрана нет. Достаточно запускать проект сочетанием клавиш Ctrl+F5, в этом случае среда вставит консольную команду задержки (pause), выполняющуюся после вашей программы. Далее в листингах не будут указываться очевидные директивы `#include` и `using namespace std`, а также `cin.get()` с целью задержки экрана.

У (1.2) Определите, что делает программа.

```
int main() {
  cout << "Enter two integers: ";
  int n = 0, m = 0;
  cin >> n >> m;
  cout << n * m << endl;
}
```

Рассмотрим пример 1.6 (см. ниже). Заметим следующее.

- Символ перевода каретки можно вписать непосредственно в строковый литерал с помощью специального синтаксиса, называемого **escape-последовательностью**. Если в строковом литерале встречается символ \, то дальше следует описание специального символа:

  `\n` — новая строка (*newline*, *linefeed*, `LF`);

  `\t` — табуляция (переход к следующей колонке);

  `\b` — возврат на одну позицию влево (*backspace*);

  `\"` — символ ";

  `\'` — символ ';

  `\\` — сам символ \;

  `\ooo` — символ с кодом $\texttt{ooo}_8$, где `o` — восьмеричная цифра (не более трех цифр);

  `\xHH` — символ с кодом $\texttt{HH}_{16}$, где `H` — шестнадцатеричная цифра.

- Символ и строка — разные типы, собственно символ задается целым числом — кодом этого символа в кодировке, выбранной системой при выполнении программы. Задать код символа можно, указав сам этот символ в одиночных кавычках: например, `'.'` — код символа «точка».

- Тип «символ» называется `char`, `cin.get()` возвращает значение как раз такого типа.

- Можно передать функции `getline` третий необязательный параметр — символ-разделитель, до которого из потока будет считываться «строчка». Например, таким образом можно считать предложение до точки.

- Если запустить программу 1.6 по F5 и ввести строчку, завершающуюся точкой, то окно закроется сразу после нажатия Enter (*почему?*).

Пример 1.6. Ввод строки (IV)

```
int main() {
  cout << "Enter a sentence:\n";
  string sentence;
  // читать текст до точки '.'
  getline(cin, sentence, '.');
  // выведем считанный текст
  cout << "You entered:\n" << sentence;
  // что осталось в буфере ввода?
  char ch = cin.get();
  cout << "\nLast character:" << ch;
}
```

Глава 2

Арифметика и логика

2.1. Системы счисления

Под системой счисления понимается способ записи чисел с помощью символов — **цифр**. Привычная система счисления — позиционная десятичная. Слово «десятичная» означает, что используются десять различных цифр. Слово «позиционная» означает, что положение цифры в записи числа имеет важное значение. Позиции цифр называются **разрядами**, пронумеруем их справа налево.

Например, число «7193» состоит из четырех разрядов.

цифра	7	1	9	3
разряд	3	2	1	0
название разряда	тысячи	сотни	десятки	единицы

Разряды удобно нумеровать, начиная не с единицы, а с нуля, что видно по следующей общей формуле, связывающей собственно число с его записью в позиционной системе счисления.

$$d_{n-1}d_{n-2}\ldots d_1d_0 = \sum_{i=0}^{n-1} d_i \cdot r^i,$$

$$7193 = 3 \cdot 10^0 + 9 \cdot 10^1 + 1 \cdot 10^2 + 7 \cdot 10^3.$$

Здесь d_i — цифра (числовое значение символа цифры), занимающая i-й разряд, r — **база** системы счисления. В традиционной десятичной системе $r = 10$.

При помощи n разрядов можно записать r^n разных целых чисел (от 0 до $r^n - 1$). Запись в позиционной системе легко обобщается на дроби: достаточно поставить разделитель целой и дробной частей (запятую или точку), указывающий нулевой разряд, и можно дописывать справа от него разряды, соответствующие отрицательным степеням базы.

2.1.1. Двоичная система

Простейшая позиционная система счисления соответствует $r = 2$. В двоичной системе счисления всего две цифры: 0 и 1. Двоичный разряд называется **бит** (***bi****nary digi****t***).

Двоичная система проще остальных реализуется «в железе». Так, если заряд ячейки памяти больше порогового уровня Q_1, считаем, что она хранит 1, если же заряд меньше порогового уровня Q_0, считаем, что она хранит 0. Таким образом, ячейка памяти, способная находиться в двух различимых состояниях, может хранить один бит. Две таких ячейки могут хранить два бита и суммарно находиться в $2^2 = 4$ разных состояниях (00, 01, 10 и 11).

У (2.1) Переведите числа 1001, 10011, 11100 и 1010 1010 из двоичной системы в десятичную.

Двоичные числа легко складывать «столбиком», учитывая перенос.

У (2.2) Сложите пары двоичных чисел:
$10010 + 11100$, $110010 + 111$, $10101 + 1100$, $11001 + 1101$.

Нетрудно выполнить и умножение «столбиком», сводящееся к сложению сдвинутых влево копий одного из множителей. Под **сдвигом влево** на n разрядов понимается дописывание справа от числа n нулей (ниже эти нули не выписаны). Что касается деления с остатком, то оно выполняется «наоборот»: надо вычитать из делимого максимально сдвинутый влево де-

литель, пока это возможно. Например, ниже получено, что $111\,0110_2 = 1101_2 \cdot 1001_2 + 1$.

	1101	×
	10011	=
+	10011	1
+	00000	0
+	10011	1
+	10011	1
=	11110111	

1110110 −	1001
1001 =	1000 +
101110 −	
1001 =	100 +
1010 −	
1001 =	1 =
1	1101

У (2.3) Вычислите произведения пар двоичных чисел: 10101×1100, 10011×1110, 11001×1010, 10110×1001.

Результат деления с остатком на степень базы r^i можно выписать, просто «разрезав» запись числа по i-му разряду:

$$d_{n-1}d_{n-2}\ldots d_i d_{i-1}\ldots d_1 d_0 = d_{n-1}d_{n-2}\ldots d_i \cdot r^i + d_{i-1}\ldots d_1 d_0.$$

Таким образом, частное записывается цифрами в разрядах $(n-1)\ldots i$, а остаток — цифрами в разрядах $(i-1)\ldots 0$.

Это свойство позволяет формализовать алгоритм выписывания числа x в заданной системе счисления. Имеем рекуррентное соотношение:

$$x = x_0,$$
$$x_{i-1} = x_i \cdot r + d_{i-1}.$$

Взяв остаток от деления x на r, получим цифру в младшем разряде. Далее, положим x равным частному деления x на r. Будем повторять эти операции, пока не получим 0 в качестве частного, выписывая остатки в соответствующие разряды записываемого числа. В качестве примера, переведем 133 из десятичной в двоичную систему: $133 = 66 \cdot 2 + \mathbf{1}$, $66 = 33 \cdot 2 + \mathbf{0}$, $33 = 16 \cdot 2 + \mathbf{1}$, $16 = 8 \cdot 2 + \mathbf{0}$, $8 = 4 \cdot 2 + \mathbf{0}$, $4 = 2 \cdot 2 + \mathbf{0}$, $2 = 1 \cdot 2 + \mathbf{0}$, $1 = 0 \cdot 2 + \mathbf{1}$.

Откуда получаем, что $133_{10} = 1000\,0101_2$.

У (2.4) Переведите числа 23, 89, 122 и 456 из десятичной системы в двоичную.

Для записи натурального числа x в r-ичной системе счисления достаточно $\lfloor \log_r x \rfloor + 1$ разрядов. Соответственно при переводе n-разрядного числа из системы с базой r в систему с базой p получаем около $n \log_p r$ разрядов. Так, при переводе из десятичной в двоичную систему количество разрядов возрастает в $\log_2 10 \approx 3$ раз. Действительно, $2^{10} = 1024 \approx 10^3$.

2.1.2. Шестнадцатеричная система

Исторически сложилось так, что минимальная отдельно адресуемая ячейка компьютерной памяти (называемая **байтом**), как правило, состоит из 8 бит. Две четырехбитные половинки байта называют **тетрадами**[1]. Тетрада может хранить $2^4 = 16$ разных значений, поэтому ее значение удобно записывать одной шестнадцатеричной цифрой. Традиционно в качестве шестнадцатеричных цифр используются 10 арабских цифр и латинские буквы A, B, C, D, E и F для недостающих цифр со значениями 10, 11, 12, 13, 14 и 15 соответственно. В свою очередь, байт может принимать $2^8 = 256 = 16^2$ разных значений. Значение байта можно записать двумя шестнадцатеричными цифрами. Для перевода из двоичной в шестнадцатеричную систему удобно разбить разряды двоичного числа на четверки и воспользоваться следующей таблицей.

0000	0	0100	4	1000	8	1100	C
0001	1	0101	5	1001	9	1101	D
0010	2	0110	6	1010	A	1110	E
0011	3	0111	7	1011	B	1111	F

У (2.5) Переведите числа из двоичной в шестнадцатеричную систему:

10 0111, 1001 0110, 10 1010 1000, 1111 0000 1010 0101.

У (2.6) Переведите числа из шестнадцатеричной в двоичную и десятичную системы: 40, 3F, 7AB, FEED.

[1] Англ. *nibble* или *nybble*.

2.2. Булевские операции

В рамках булевской алгебры[2] можно оперировать логическими формулами, построенными с помощью операций *не*, *и*, *или*, которые связывают переменные, принимающие значения из множества {*ложь*, *истина*}.

У (2.7) Составьте таблицу значений формулы «*не* (*A и* (*B или C*))» для всех возможных комбинаций значений A, B и C.

Помимо операций *и* и *или* часто применяется операция *исключающее или* («или-или»), ее можно выразить как *A иск. или B* = (*не A и B*) *или* (*A и не B*). Таким образом, *исключающее или* исключает случай, когда оба операнда истинны. Нетрудно заметить, что *A иск. или B* истинно тогда и только тогда, когда $A \neq B$.

операция	названия	англ.	обозначения	C++, логика	C++, побит.
не A	отрицание, дополнение	not	$\neg A$, $\overline{A}$	`!A`	`~A`
A и B	конъюнкция, умножение	and	$A \wedge B$, $A \cdot B$	`A && B`	`A & B`
A или B	дизъюнкция, сложение	or	$A \vee B$, $A + B$	`A || B`	`A | B`
A иск. или B	строгая дизъюнкция, сложение по модулю 2	xor, eor	$A \veebar B$, $A \oplus B$	`A != B` `!A!=!B`	`A ^ B`

В работах по логике *ложь* обычно обозначают через «0», а *истину* — через «1». Ниже приведены **таблицы истинности**, с помощью которых можно задавать операции *и*, *или* и *исключающее или*.

[2]Существует множество булевских алгебр, мы ограничимся лишь простейшим нетривиальным случаем.

$\wedge$	0	1
0	0	0
1	0	1

$\vee$	0	1
0	0	1
1	1	1

$\oplus$	0	1
0	0	1
1	1	0

У (2.8) Составьте таблицы истинности операций.

1. *Стрелка Пирса* (*или-не*, nor) $A \downarrow B = \neg(A \vee B)$, «ни A, ни B».
2. *Штрих Шеффера* (*и-не*, nand) $A \uparrow B = \neg(A \wedge B)$.
3. *Эквиваленция* (nxor, eqv) $A \leftrightarrow B = \neg(A \oplus B)$, «$A = B$».
4. *Импликация* $A \to B = \neg A \vee B$, «$A \leqslant B$».

У (2.9) Докажите, что через *или-не* можно выразить операции *не*, *и*, *или*.

У (2.10) Докажите, что через *и-не* можно выразить операции *не*, *и*, *или*.

Булевский тип в C++ задается ключевым словом `bool`. Константы *истина* и *ложь* — ключевыми словами `true` и `false` соответственно. Ниже показан листинг 2.1, позволяющий вычислить значение булевской формулы $(A \vee B) \wedge \neg C$ для одного частного случая значений переменных.

Пример 2.1. Вычисление логической формулы (I)

```
// булевские переменные — истина или ложь
bool A = true, B = false, C = true;
// вычислим значение формулы
bool formula = (A || B) && !C;
```

Булевские значения можно вводить с клавиатуры и выводить на экран (по умолчанию используются 0 и 1).

Пример 2.2. Вычисление логической формулы (II)

```
// булевские переменные — истина или ложь
bool A, B, C;
cin >> A >> B >> C;
// вычислим и выведем значение формулы
cout << ((A || B) && !C);
```

У (2.11) Перепишите пример 2.2 для следующих формул.

1. $(A \wedge \neg B) \oplus (\neg C \vee D)$.
2. $(A \vee B) \wedge \neg(C \oplus D)$.
3. $(A \oplus \neg B) \vee (\neg C \wedge D)$.
4. $(A \vee B \wedge \neg C) \oplus D$.

В программах булевские значения, как правило, существуют не сами по себе, а возникают в ходе проверки условий. Например, при вычислении выражений, содержащих операции сравнения.

условие	в C++	условие	в C++	условие	в C++
$a = b$	`a == b`	$a < b$	`a < b`	$a \leqslant b$	`a <= b`
$a \neq b$	`a != b`	$a > b$	`a > b`	$a \geqslant b$	`a >= b`

Внимание! В C++ оператор присваивания `=` записывает значение, указанное справа, в переменную, указанную слева. Для сравнения на равенство используется операция `==`.

Итак, допустим, пользователь вводит n и m, а наша программа должна сообщать ему, верно ли, что $(2n < 3m) \wedge (n \neq m)$. Запишем это условие на C++.

Пример 2.3. Использование операторов сравнения

```
// целые числа n и m
int n, m;
cin >> n >> m;
// вычислим и выведем значение формулы
cout << (2 * n < 3 * m && n != m);
```

У (2.12) Перепишите пример 2.3 для следующих формул трех целочисленных переменных.

1. $(n^2 + m^2 < p^2) \wedge (n + m + p > 0)$.
2. $((n - m)^2 + (m - p)^2 < p^2) \vee (p < m + n)$.

3. $(nm < np) \wedge (n + m \leqslant p + 1)$.

4. $(n = m \vee n = p \vee m = p) \oplus (n \neq m \wedge 4n \geqslant p)$.

2.3. Поразрядные операции

Поразрядными булевскими операциями (**побитовыми операциями**) называются операции, предполагающие применение булевских операций к каждой паре двоичных разрядов, стоящих в одной позиции, независимо от других разрядов. Например, *поразрядное и*, обозначаемое в C++ через `&`, вычисляет булевское *и* бит, стоящих в числах слева и справа на одной позиции. Аналогично можно ввести *поразрядное или* `|` и *поразрядное исключающее или* `^`. Доступно и поразрядное отрицание (обычно называемое *дополнением* или *инверсией*), которое обозначается `~`.

1100	`&`	1100	`|`	1100	`^`	`~` 1100	=
1010	=	1010	=	1010	=	0011	
1000		1110		0110			

Операция *поразрядное исключающее или* обладает следующими свойствами.

1. Коммутативность: `a ^ b` $\equiv$ `b ^ a`.

2. Ассоциативность: `(a ^ b) ^ c` $\equiv$ `a ^ (b ^ c)`.

3. Управляемое отрицание: `0 ^ a` $\equiv$ `a`, `~0 ^ a` $\equiv$ `~a`.

4. `(a ^ b) ^ a` $\equiv$ `b`, в частности, `a ^ a` $\equiv$ `0`.

Кроме булевских поразрядных операций, применяются **операции сдвига бит**, заключающиеся в «сдвиге» значений влево или вправо по разрядам. При этом разряды, «выдвигаемые» за пределы фиксированного машинного слова, могут отбрасываться (*линейный сдвиг*) или «задвигаться» с противоположной стороны (*циклический сдвиг*). В случае линейного сдвига

влево освобождающиеся правые (*младшие*) разряды заполняются нулями. В случае линейного сдвига вправо возможно заполнение освобождающихся левых (*старших*) разрядов строго нулями либо значением самого старшего разряда[3] (такой сдвиг называется *арифметическим*, потому что старший разряд равен нулю, если число неотрицательное, и равен единице, если число отрицательное). Ниже показаны результаты сдвига 8-битного слова на 0–4 бит вправо для циклического, линейного и арифметического сдвигов.

циклич.	линейн.	арифм.	бит
11001101	11001101	11001101	0
11100110	01100110	11100110	1
01110011	00110011	11110011	2
10111001	00011001	11111001	3
11011100	00001100	11111100	4

Маской называют последовательность бит, задающую, значения каких бит 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 нужных местах. При наложении такого шаблона на текст (вычислении поразрядного u) мы видим только те буквы, которые попали под отверстия (биты в тех позициях, где в маске стоят единицы).

Поразрядные операции относятся к простейшим операциям с данными, которые способен выполнять процессор, поэтому ими нередко заменяют вычислительно тяжелые арифметические действия (предполагаем, что результаты представимы в машинном слове), например:

- степень 2^n можно получить как `1u << n` (линейный сдвиг влево), соответственно произведение $2^n a$ можно получить как `a << n`;

[3]Предполагается, что отрицательные числа задаются в дополнительном коде.

- целая часть частного $\frac{a}{2^n}$ при условии $a \geqslant 0$ соответствует `a >> n` (линейный сдвиг вправо);

- остаток от деления $a \geqslant 0$ на 2^n можно получить, выделив младшие биты (которые пропали бы при сдвиге вправо) с помощью соответствующей маски: `a & ((1u << n) - 1)`. В частности, можно проверить число a (справедливо и для отрицательных в дополнительном коде) на четность с помощью условия `a & 1 == 0`.

В языках C и C++ важную роль играют понятия **неопределенное поведение** (*undefined behaviour*, UB) и **определяемое реализацией поведение** (*implementation-defined behaviour*, IB). Когда некоторое действие объявляется как порождающее UB, это означает, что программист не должен ожидать какой-то определенный результат — все зависит от выбора компилятора в данном конкретном случае и особенностей платформы, причем поставщики не обязаны указывать в документации последствия такого действия. В случае IB, поставщик 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. Ярким примером UB и ошибочного кода является повторное использование (в том числе повторное изменение) изменяемой переменной при вычислении выражения, когда относительный порядок вычисления термов не определен (то же касается фактических параметров функции в точке вызова):

```
++a -= b;                          // UB
cout << a << a++ << ++a;           // UB
arr[a = 10] = a;                   // UB
arr[0] = 0;
arr[arr[0]] = 1;                   // UB
```

Впрочем, в ряде случаев допустимо использовать код, объявленный в стандарте как вызывающий UB или IB, так как

определенные гарантии предоставляются основными платформами. Например, вызывающим UB объявлено любое арифметическое действие с целыми числами, результат которого не может быть представлен в используемом машинном представлении числа. Однако подавляющее число процессоров общего назначения реализуют арифметику с оборачиванием (то есть отбрасыванием старших разрядов, не поместившихся в представлении), а отрицательные числа представляют в дополнительном коде, что делает поведение предсказуемым в большинстве случаев[4]. Но даже здесь имеются нюансы. Например, целочисленное деление на ноль и умножение или деление наименьшего представимого отрицательного целого числа на -1 по-разному обрабатываются на разных платформах.

Что до операций сдвига, то поведение не определено, когда правый операнд меньше нуля или не меньше числа бит в представлении либо левый операнд является `signed` и результат сдвига влево не может быть представлен как произведение левого операнда на степень двойки. Смысл сдвига вправо отрицательных значений определяется реализацией (как правило, это арифметический сдвиг).

У (2.13) Напишите программу, запрашивающую целое число и выводящую частное и остаток от деления на 16 с помощью операций сдвига и поразрядного u, а также с помощью обычных операций `/` и `%`. (Всего выводит четыре числа.) Попробуйте ввести отрицательное число, проанализируйте полученный результат.

У (2.14) IPv4-адрес имеет размер 32 бита и может быть разделен на две составляющие: адрес сети (старшие биты) и адрес машины (младшие биты). Сам адрес обычно записывается десятичными значениями четырех байт, разделенных точками: например, 192.168.10.12. Количество бит, задающих адрес сети, записывают через косую черту: 192.168.10.12/16 — все маши-

[4] Тем не менее следует проявлять осторожность, по возможности избегая подобных ситуаций, сверяться с документацией и проверять результирующий код.

ны в этой сети имеют адреса, начинающиеся на 192.168. Чтобы извлечь адрес сети $/n$, следует наложить маску с установленными битами в позициях $31 \dots (32 - n)$. Чтобы извлечь адрес машины, следует наложить дополнение этой маски.

Напишите программу, которая вначале считывает IPv4-адрес (как четыре целых числа) и размер маски и выводит маску сети, адрес сети и адрес машины, затем считывает другой IPv4-адрес и заменяет в нем: а) адрес сети на адрес сети из первого адреса; б) адрес машины на адрес машины из первого адреса (вывести два получившихся адреса).

Пример. Определить четность числа установленных бит.

Рассмотрим 8-битное число $\mathrm{b} = b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$. Число установленных бит можно получить, сложив все разряды числа: $\sum_{i=0}^{7} b_i$, четность равна остатку от деления суммы на 2. Иными словами, вместо сложения отдельных бит можно использовать операцию *исключающее или* (сложение по модулю 2).

Обратим внимание, что благодаря ассоциативности данной операции можно «складывать» биты в произвольном порядке, например, так: $((b_7 \oplus b_6) \oplus (b_5 \oplus b_4)) \oplus ((b_3 \oplus b_2) \oplus (b_1 \oplus b_0))$, то есть сложить пары соседних бит, потом соседние пары бит результата (из которых значим только младший бит), потом соседние четверки бит и т. д. Данный метод применим к произвольному числу бит и позволяет ускорить вычисление с помощью применения поразрядных операций, одновременно задействующих группы бит, уложенные в одном машинном слове.

```
// b содержит 32 бита
b ^= b >> 1; // соседние биты
b ^= b >> 2; // соседние пары бит
b ^= b >> 4; // соседние четверки бит
b ^= b >> 8; // соседние байты
b ^= b >> 16; // соседние пары байт
int parity = b & 1; // четность в младшем бите
```

Получить сумму младших бит во всех четверках бит числа можно одной операцией умножения на число, составленное

из четверок бит 0001_2, длиной в машинное слово (*почему?*), при этом результат будет находиться в четырех старших битах. Таким образом, вышеприведенный код можно заменить на следующий код.

```
// b содержит 32 бита
b ^= b >> 1; // соседние биты
b ^= b >> 2; // соседние пары бит
b = (b & 0x11111111) * 0x11111111;
int parity = (b >> 28) & 1;
```

У (2.15) Реализовать циклический сдвиг влево и вправо, используя линейный сдвиг и поразрядное *или*.

У (2.16) Проверить, является ли целое число степенью двойки. (*Подсказка*: вычесть единицу.)

У (2.17) Установить в числе младший нулевой бит. Сбросить в числе младший ненулевой бит.

У (2.18) Сбросить в числе все биты, кроме младшего ненулевого бита.

У (2.19) Обменять старшую и младшую половины двоичного числа местами.

У (2.20) Обратить порядок байт в числе.

У (2.21) Обратить порядок бит в числе.

У (2.22) Посчитать число установленных бит.

Отдельные биты могут использоваться как признаки наличия элементов из некоторого конечного множества в подмножестве (представленном последовательностью бит). Таким образом, если есть множество $B = \{0, 1, \ldots 7\}$, то 8-битный байт может служить представлением произвольного подмножества $M \subseteq B$. Если $i \in M$, то i-й бит байта установлен в единицу. Указанный способ представления подмножеств позволяет выполнять теоретико-множественные операции с помощью поразрядных операций.

У (2.23) В таблице ниже заменить знаки «?» на соответствующую C++-запись.

операция		C++
принадлежность	$i \in A$	`(1u << i) & A != 0`
подмножество	$A \subseteq B$	?
дополнение	$\overline{A}$	`~A`
пересечение	$A \cap B$	`A & B`
объединение	$A \cup B$	?
разность	$A \setminus B$	?
симметр. разность	$A \triangle B$	?
пустое множество	$\varnothing$	`0u`
полное множество		?

Глава 3

Функции

3.1. Функциональная декомпозиция

Вначале мы имели дело лишь с одной функцией `main`, которая содержала основную часть программы, не являясь частью чего-то большего. Однако предназначение функции в C++ — решать некоторую подзадачу в рамках библиотеки программных компонент или приложения (программного комплекса, отвечающего определенному кругу задач в рамках заданной предметной области и предоставляющего пользовательский интерфейс). Таким образом, «функция» в C++ — это не функция в математическом смысле, а вид процедуры, то есть стандартным образом оформленный фрагмент кода, имеющий имя, способный принимать набор значений (параметров) и по завершении возвращать полученный результат.

Рассмотрим простую программу, запрашивающую число и выводящую его квадрат.

```
int main() {
  cout << "Enter a number: ";
  // n — целочисленная переменная
  int n = 0;
  // считаем из cin целое число в переменную n
  cin >> n;
```

```
  // выведем квадрат n
  cout << n << " squared is " << (n * n);
}
```

Если требуется вычислять квадрат выражения, то разумно определить функцию «квадрат». Следующая программа запрашивает n и вычисляет $(n^2 - 1)^2$.

```
// функция, вычисляющая квадрат аргумента
int square(int n)
{ return n * n; }

int main() {
  cout << "Enter a number: ";
  // n — целочисленная переменная
  int n = 0;
  // считаем из cin целое число в переменную n
  cin >> n;
  // выведем значение формулы
  cout << "(n^2 - 1)^2 = "
       << square(square(n) - 1);
}
```

Представьте, как могла выглядеть эта формула, если бы мы записали ее с помощью операций произведения. Строчка

```
int square(int n)
```

является **заголовком функции** и состоит из трех частей: *типа возвращаемого значения* (здесь — `int`), *названия функции* (`square`), с помощью которого к ней можно обращаться в программе, и, наконец, *списка параметров*, размещаемого в круглых скобках (скобки ставятся даже в случае отсутствия параметров), параметры оформляются парами *тип-параметра имя-параметра* и разделяются запятыми.

Если после заголовка функции поставить точку с запятой, то получим **объявление функции**, описывающее, как вызывать эту функцию (что она принимает и возвращает), но не определяющее ее (что она делает).

Если после заголовка функции следует **тело функции**, заключенное в фигурные скобки `{}`, то получим **определение функции**, которое полностью определяет функцию. В нашем случае тело функции уместилось в одну строчку

```
{ return n * n; }
```

и состоит из инструкции `return`, возвращающей из функции значение (здесь квадрат `n`) коду, вызвавшему эту функцию.

Отдельные функции разумно вводить также в том случае, если в разных местах программы требуется выполнять одинаковый код, в частности вычислять одну и ту же формулу с разными параметрами.

В целом процесс решения некоторой задачи можно представить как комбинацию двух основных действий: анализа и синтеза.

Анализ заключается в изучении задачи и разложении ее на составляющие элементы (понятия, объекты, правила и подзадачи). В частности, разбиение решения задачи в последовательность решений подзадач, которые можно представить в виде отдельных действий, позволяет сформулировать нужный алгоритм и называется **функциональной декомпозицией**.

Синтез заключается в подборе подходящих элементов из набора готовых компонент и составлении из них нужных конструкций. Набор готовых компонент (будь то фрагменты кода, литературные обороты, решения классов математических задач или что-то еще) может пополняться за счет повторения действий анализа и синтеза.

В рамках процедурного программирования состояния объектов задачи задаются в виде наборов переменных, а алгоритмы в виде явных последовательностей инструкций, изменяющих состояния объектов, и требуется разбивать алгоритм на элементы до тех пор, пока не станет возможным выразить их все, используя имеющиеся конструкции языка программирования. Основной сложностью при разработке программного обеспечения является высокий (часто абсолютно не осознаваемый ни заказчиками, ни разработчиками) уровень неопределенно-

сти, которую требуется снять, чтобы, во-первых, довести детализацию анализа до такого уровня, что можно приступать к программированию, и, во-вторых, суметь реализовать программу, удовлетворяющую заданным условиям.

3.2. Вычисление значений формул

Изучите программу 3.1. Каковы ее недостатки?

Пример 3.1. Решение линейного уравнения (I)

```
// функция, решающая уравнение ax + b = 0
double lineq(double a, double b)
{ return -b / a; }

int main() {
  cout << "Enter a, b: ";
  double a = 1.0, b = 0.0;
  cin >> a >> b;
  cout << a << "x + " << b
       << " = 0, x = " << lineq(a, b);
}
```

У **Задание**

1. Запросить у пользователя значения параметров a, b, c.
2. Вывести решение x уравнения $f(x) = 0$ для заданных параметров.
3. Выполнить проверку путем вычисления $f(x)$ от полученного x (вывести на экран).

Считать, что для заданных значений параметров решение существует. При наличии более одного решения выводить любое. Оформить вычисление решения уравнения в виде функции, принимающей параметры a, b и c.

Варианты

(3.1) $f(x) = a\sqrt{b + 8c\sqrt{x}} - bc$

(3.2) $f(x) = \dfrac{b}{a - x} - \dfrac{3ac}{x^2}$

(3.3) $f(x) = \dfrac{ax^7}{b} + cx^{-2}$

(3.4) $f(x) = 2c - \dfrac{x^4 - a^4}{b + c}$

(3.5) $f(x) = 5c - \ln(ax - b)$

(3.6) $f(x) = c\ln(4a^2x^2) + b$

(3.7) $f(x) = 1 - \dfrac{b}{c}\exp\left(a + \dfrac{2x}{c}\right)$

(3.8) $f(x) = 1 - \dfrac{a^2}{b^2 - c^2}e^{-\frac{x}{c}}$

(3.9) $f(x) = b + (2a)^{2x+c}$

(3.10) $f(x) = 8a - \left(\dfrac{ab}{c^3}\right)^{-x}$

(3.11) $f(x) = |b|c - \sqrt{4x^2 + a^2}$

(3.12) $f(x) = ||ax| - |b|^c|$

(3.13) $f(x) = \sin\left(\dfrac{a^2}{c^2}|x| + 3b\right)$

(3.14) $f(x) = ab - \cos(a\sqrt{c - x})$

(3.15) $f(x) = 2a - b\arcsin(cx)$

(3.16) $f(x) = b^2 - \arccos\dfrac{2a}{c - x}$

(3.17) $f(x) = a\,\mathrm{tg}\,(b^2x^2 - a^2) - c$

(3.18) $f(x) = 1 + \dfrac{\sin(ax - b)}{\cos(ac - b)}$

(3.19) $f(x) = 1 - 3bc + \mathrm{arctg}\,(3ax + 100^{-c})$

(3.20) $f(x) = |b + \sin\pi c| - \dfrac{2^a}{x^2}$

Пример решения для заданий 3.1–3.20. Дано:

$$f(x) = 1 + \sin\left(a^x + |\log_b c|\right).$$

Решение

Положим $f(x) = 0$ и выразим x, формально удовлетворяющее этому уравнению: $1 + \sin\left(a^x + |\log_b c|\right) = 0 \Rightarrow a^x + |\log_b c| = -\frac{\pi}{2} + 2\pi\mathbb{Z} \Rightarrow a^x = -|\log_b c| - \frac{\pi}{2} + 2\pi\mathbb{Z} \Rightarrow x = \log_a\left(-|\log_b c| - \frac{\pi}{2} + 2\pi\mathbb{Z}\right)$.

Так как возможных решений потенциально бесконечно много, дозволяется зафиксировать константу (так, чтобы для некоторых a, b и c полученная формула имела значение) из $\mathbb{Z}$. Например, взяв 1, формально получим

$$x = \log_a\left(\frac{3\pi}{2} - |\log_b c|\right).$$

Запишем полученное решение на языке C++.

Пример 3.2. Вычисление значений формул

```
#include <cmath>
// константа — половина пи
const float HALF_PI = 1.570796326795;
// вычислим значение f для заданных значений
// переменной x и параметров a, b и c
float f(float x, float a, float b, float c) {
  return 1.0 + sin(pow(a, x)
             + abs(log(c) / log(b)));
}
// вычислим значение x (корня f(x) = 0) для
// заданных значений параметров a, b и c
float root(float a, float b, float c) {
  return log(3.0 * HALF_PI
         - abs(log(c)/log(b))) / log(a);
}
```

```
int main() {
  // запросить значения параметров a, b и c
  cout << "enter a b c\n";
  float a, b, c;
  cin >> a >> b >> c;
  // найти решение уравнения f(x) = 0
  const float x = root(a, b, c);
  cout << "x = " << x;
  // проверить найденное решение подстановкой
  cout << "\nf(x) = " << f(x, a, b, c);
}
```

3.3. Проверка условий

```
if (some_condition)
  some_instruction;
```

Инструкция `some_instruction` выполнится, если значение выражения `some_condition` приводится к значению *истина* (в частности, целые числа и адреса приводятся к истине, если не равны нулю). В противном случае `some_instruction` не выполнится.

Возможен и полный вариант инструкции ветвления, предлагающий альтернативное действие для случая, когда условие приводится к значению *ложь*.

```
if (some_condition)
  instruction_when_true;
else
  instruction_when_false;
```

Если требуется выполнить несколько инструкций по условию, то их следует сгруппировать в **блок**, поместив между парой фигурных скобок.

```
if (a < 10) {
  cout << "a is less than 10\n";
```

```
  cout << "this is the second instruction!\n";
  a = 10; // пусть a = max(a, 10)
}
```

Блок может существовать и «сам по себе», независимо от других инструкций. Блок открывает собственную область видимости для локальных определений.

```
int a = 10;
{
  int a = 42; // локальная a
  cout << a << endl; // 42
}
cout << a << endl; // 10
```

В качестве условия может выступать любая булевская формула. Операции *и* (`&&`) и *или* (`||`) вычисляются по «короткой схеме»: правый операнд не вычисляется, если значение левого операнда уже однозначно определяет значение выражения (если слева от *и* — *ложь*, то независимо от значения правого операнда, все равно получим *ложь*, аналогично для *или* и *истина*). Все побочные эффекты, связанные с вычислением левого операнда *и* и *или*, вступают в силу до вычисления правого операнда.

```
if (a && b) // если a истинно, проверить b
  do_smth; // выполнить, если и b истинно
if (a || b) // если a ложно, проверить b
  do_smth; // выполнить, если a или b истинно
```

Под `if` и `else` может находиться другая инструкция, в том числе `if`. Это позволяет написать «каскадный» `if-else` для случая выбора из более чем двух возможных ситуаций.

```
if (condition_1)
  action_1; // если condition_1 истинно
else if (condition_2) // condition_1 ложно,
  action_2; // а condition_2 истинно
else if (condition_3) // condition_1 и _2 ложны,
```

```
  action_3; // a condition_3 истинно
else
  action; // все три условия ложны
```

Вычисление операндов по условию можно встроить в выражение с помощью тернарного оператора `?:`. Например, функцию *максимум двух чисел* можно записать так:

```
double max(double a, double b) {
  if (a < b) return b;
  return a;
}
```

Однако с помощью `?:` то же самое можно записать короче:

```
double max(double a, double b)
{ return a < b? b: a; }
```

У (3.21) Используя тернарный оператор, определите следующие функции.

1. *Минимум двух чисел* `min`.

2. *Модуль числа* `abs`.

3. *Знак числа* `sgn`.

Вернемся к примеру 3.1. Заметим, что при попытке ввести $a = 0$ или даже $a = 0$, $b = 0$ мы не получим корректного ответа. Переделаем пример 3.1 так, чтобы функция возвращала количество корней (0, 1 или «бесконечность»), а сам корень записывала в переменную, **адрес** которой передан в качестве дополнительного параметра. Адреса значений типа `T` в языке C++ называются **указателями** на `T`.

тип «указатель на `T`»	→	`T*`
обращение к значению по адресу `p`	→	`*p`
взятие адреса переменной `x`	→	`&x`

Пример 3.3. Решение линейного уравнения (II)

```
// условное значение "бесконечное число"
const int INFINITY = -1;
// функция, вычисляющая решение ax + b = 0
int lineq(double a, double b, double *root) {
  if (a == 0.0) { // 0x + b = 0 имеет либо
    // бесконечное число корней, либо ни одного
    return b == 0.0? INFINITY: 0;
  }
  // a != 0, следовательно, есть один корень
  *root = -b / a;
  return 1;
}

int main() {
  cout << "Enter a, b: ";
  double a = 1.0, b = 0.0;
  cin >> a >> b;
  cout << a << "x + " << b << " = 0, ";
  double root;
  // рассмотрим различные варианты
  switch (lineq(a, b, &root)) {
  case 0: cout << "has no roots\n"; break;
  case 1: cout << "x = " << root << '\n'; break;
  default: cout << "many roots\n";
  }
}
```

У (3.22) Используя пример 3.3, написать функцию, решающую квадратное уравнение. Эта функция должна вызывать функцию `lineq` в случае равенства нулю коэффициента при x^2.

У (3.23) Взять решение задания (3.1–3.20) и дополнить его:

- проверкой на существование решения;
- (если это имеет смысл для данного уравнения) проверкой на удовлетворение уравнения почти всеми $x \in \mathbb{R}$.

При выполнении задания достаточно ограничиться рассмотрением одного из возможных корней в случаях, аналогичных рассмотренному в примере ниже. Проверки оформить в виде отдельных функций, принимающих параметры уравнения a, b и c и возвращающих булевское значение.

Пример. Дано

$$1 + \sin\left(a^x + |\log_b c|\right) = 0,$$

откуда формально получено (один из возможных корней)

$$x = \log_a \left(\frac{3\pi}{2} - |\log_b c| \right).$$

Данная формула не имеет смысла при $a < 0$, $b \leqslant 0$ или $c \leqslant 0$. Отдельного рассмотрения заслуживают случаи $a = 0$ и $a = 1$, так как при этих значениях a, возможно, подойдет почти любое $x \in \mathbb{R}$. Для этого необходимо, чтобы выполнялось, например (для $x = 1$):

$$1 + \sin\left(a + |\log_b c|\right) = 0.$$

Взяв в качестве основы пример 3.3, приведенные выше рассуждения можно выразить на C++ следующим образом (общая с решением предыдущего примера часть кода опущена).

Пример 3.4. Проверка существования решений

```
// допустимое отклонение от нуля
const float TOLERANCE = 1e-6;

int root(float a, float b, float c, float *x) {
  if (a < 0.0 || b <= 0.0 || c <= 0.0) return 0;
  if (a == 0.0 || a == 1.0)
    return abs(f(1.0, a, b, c)) <= TOLERANCE?
                INFINITY: 0;
  *x = log(3.0 * HALF_PI
       - abs(log(c)/log(b))) / log(a);
```

```
  return 1;
}

int main() {
  // получить значения a, b и c
  cout << "enter a b c\n";
  float a, b, c, x;
  cin >> a >> b >> c;
  // рассмотрим различные варианты
  switch (root(a, b, c, &x)) {
  case 0: cout << "has no roots\n"; break;
  case 1:
    cout << "x = " << x << '\n';
    // проверить найденное решение подстановкой
    cout << "\nf(x) = " << f(x, a, b, c);
    break;
  default: cout << "many roots\n";
  }
}
```

В C++ функция может ничего не возвращать (в других языках, например в Pascal, такая конструкция может называться *процедурой*), для этого в качестве типа возвращаемого значения следует указать `void`.

```
#include <cstdlib> // std::system
void cls()
{ system("cls"); }

int main() {
  cout << "On Windows you ain't gonna see it!";
  cls(); // вызвать процедуру cls
}
```

Функция `cls()` выполняет некие действия и завершается, ничего не возвращая. Напишем функцию, которая считывает и выводит числа, пока пользователь не закроет окно.

```
void enter_ints() {
  while (true) {
    int n = 42;
    cin >> n;
    cout << "you've entered " << n << endl;
  }
}
```

Цикл `while` повторяет расположенный под ним код (блок или инструкцию), пока истинно условие, указанное в круглых скобках сразу за ключевым словом `while`. Если условие всегда истинно, то цикл называется **бесконечным**.

Попробуйте вместо числа ввести букву. Возникнет ошибка ввода, символы, не являющиеся цифрами, прочитаны не будут, а значение переменной `n` не изменится (так как операция чтения не завершилась). Аналогичный эффект в этой программе возникнет, если ввести символ *признака конца файла* (*end of file*, *eof*). При работе в консоли, признак конца файла вводится через Ctrl+Z (Windows) или Ctrl+D (Unix). После ввода признака конца файла поток ввода «отключается» от консоли и при попытке прочитать символ возвращает один и тот же специальный код `EOF` (обычно равен -1).

Вызов `cin.eof()` позволяет узнать, встретился ли конец файла. Вызов `cin.fail()` позволяет проверить, произошла ли ошибка ввода. Вызов `cin.clear()` сбрасывает состояние потока. Вызов `cin.ignore()` убирает из потока следующий символ, если он есть. (В случае Visual Studio можно использовать `cin.sync()` для того, чтобы сбросить буфер ввода.)

Используем эти функции, чтобы проверить успешность считывания числа и корректно обработать ошибки (будем игнорировать ошибки ввода и прервем цикл при поступлении признака конца файла).

Пример 3.5. Цикл чтения из потока

```
void enter_ints() {
  while (true) {
```

```
    int n = 42;
    cin >> n;
    if (cin.eof()) {
      cin.clear();  // сбросить состояние потока
      break; // выйти из цикла
    }
    if (cin.fail()) {
      cin.clear();
      cin.ignore(); // отбросить символ
    }
    else // успешно ввели число, обработать его
      cout << "you've entered " << n << endl;
  }
}
```

У **Задание** (рис. 3.1–3.5). Дана фигура, образованная комбинацией кругов и полуплоскостей (используются прямоугольники со сторонами, параллельными осям, и «ромбы» — квадраты, повернутые на 45°). Требуется написать функцию, проверяющую попадание точки с заданными координатами в эту фигуру.

Программа должна позволять проверить произвольное количество точек за один запуск и корректно обрабатывать ошибки ввода. Оформить проверку попадания как отдельную функцию.

Пример. Фигура на рис. 3.6 является пересечением полуплоскости $x \geqslant -4$ и объединения следующих фигур:

- прямоугольник с углами $(-4, -5)$ и $(-2, 5)$;
- прямоугольник с углами $(2, -5)$ и $(4, 5)$;
- круг радиуса 5 с центром в точке $(-2, 0)$.

Пример 3.6. Проверка попадания точки в фигуру

```
// проверить попадание в прямоугольник
bool inRect(float x, float y,
```

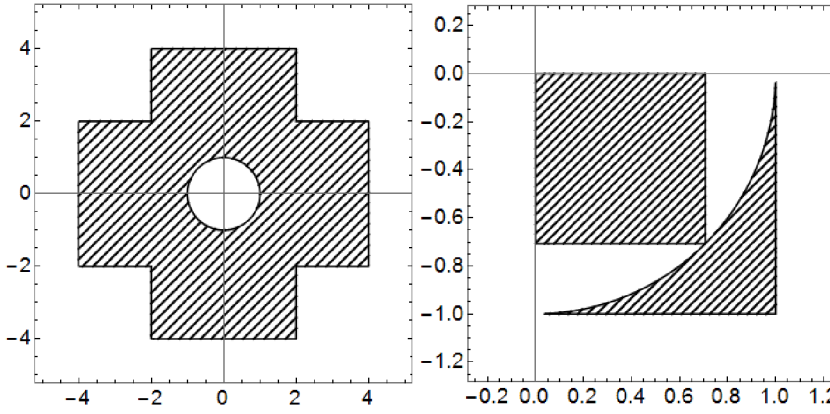

Рис. 3.1. Варианты (3.24) и (3.25)

```
                float left, float right,
                float bottom, float top) {
  return left <= x && x <= right
    && bottom <= y && y <= top;
}
// проверить попадание в круг
bool inCircle(float x, float y,
        float cx, float cy, float r) {
  float dx = x - cx, dy = y - cy;
  return dx * dx + dy * dy <= r * r;
}
// проверить попадание в заданную фигуру
bool inFigure(float x, float y) {
  return inRect (x, y,  2.0,  4.0, -5.0, 5.0)
      || inRect (x, y, -4.0, -2.0, -5.0, 5.0)
      || (inCircle (x, y, -2.0, 0.0, 5.0)
          && x >= -4.0);
}
int main() {
  // проверять точки, пока не встретится
  // признак конца файла
```

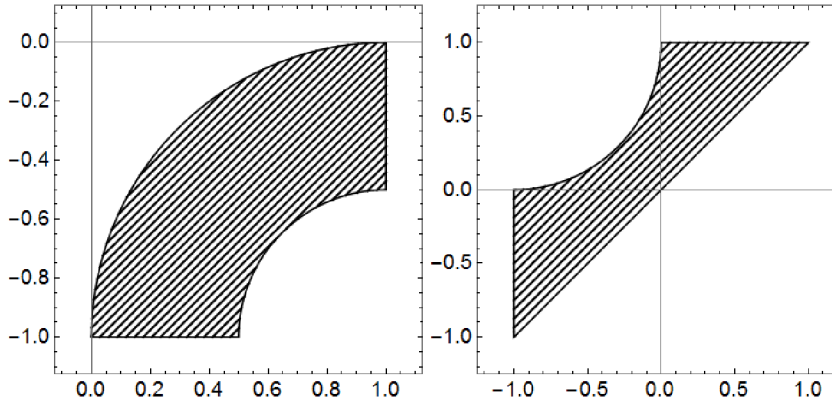

Рис. 3.2. Варианты (3.26) и (3.27)

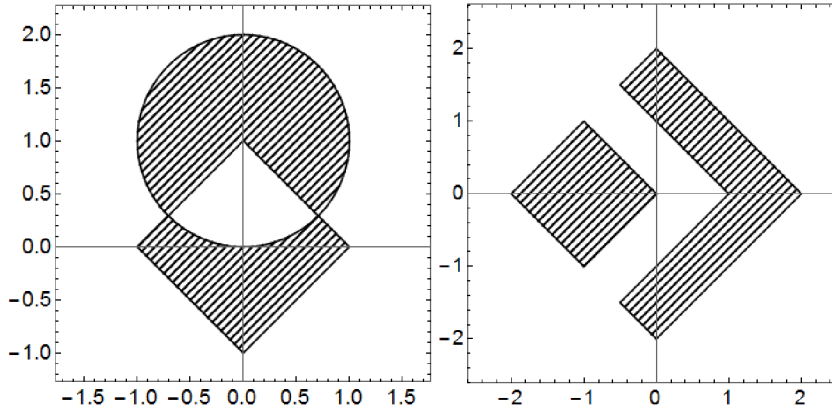

Рис. 3.3. Варианты (3.28) и (3.29)

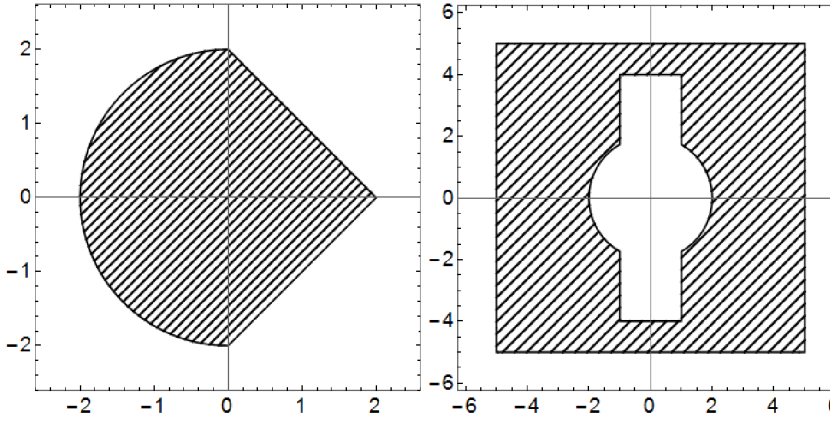

Рис. 3.4. Варианты (3.30) и (3.31)

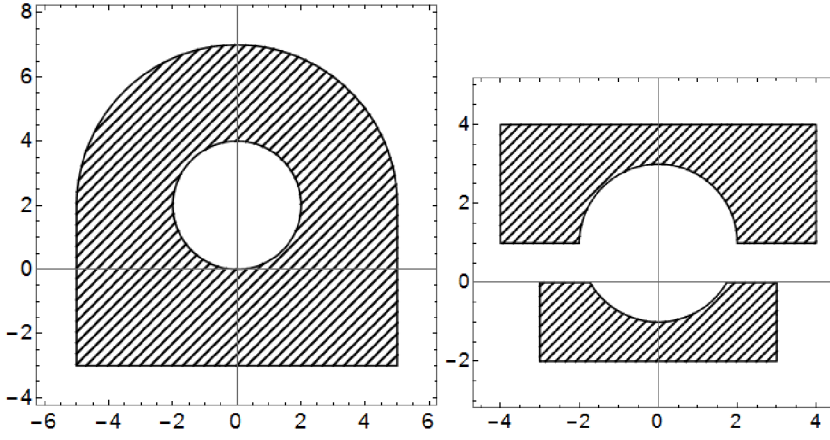

Рис. 3.5. Варианты (3.32) и (3.33)

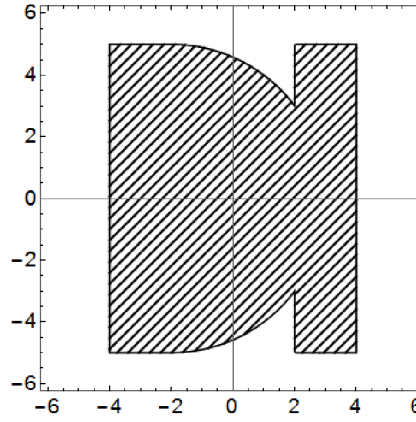

Рис. 3.6. Пример

```
  while (true) {
    // координаты проверяемой точки
    float x, y;
    cin >> x >> y;
    // проверить успешность ввода
    if (cin.eof()) {
      cin.clear(); // сбросить ошибки потока
      break; // выйти из цикла
    }
    if (cin.fail()) {
      cin.clear();
      cin.ignore(); // отбросить символ
    }
    else // проверить попадание
      cout << inFigure(x, y) << "\n";
  }
}
```

3.4. Рекурсия

Рекурсией называется прямой или косвенный вызов функцией себя самой.

Пример 3.7. Факториал

```
double fact(int n)
{ return n < 2? 1.0: n * fact(n); }
```

У (3.34) Написать рекурсивную функцию, выводящую двоичное представление числа.

Числами Фибоначчи называется последовательность чисел F_n, заданная рекуррентным соотношением

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}.$$

У (3.35) Написать функцию `fib(int)`, вычисляющую число Фибоначчи заданного номера рекурсивно, используя приведенное выше определение. Попробуйте с помощью нее вычислить F_{100}. Какова причина такого результата?

Рекурсивное вычисление чисел Фибоначчи «сверху-вниз» можно преобразовать в вычисление «снизу-вверх», так, как если бы человеку пришлось вычислять заданное число Фибоначчи на бумаге. Для вычисления следующего числа требуется знать только два предыдущих. Это можно выполнить, передавая функции значения «сколько еще чисел надо посчитать», «предыдущее число» и «предпредыдущее число».

Пример 3.8. Числа Фибоначчи «снизу-вверх»

```
double fib(int n, double prev1, double prev2) {
  return n < 1? prev2: fib(n - 1,
                           prev1 + prev2, prev1);
}
// используем перегрузку для удобства вызова
double fib(int n) {
  return fib(n, 1.0, 0.0);
}
```

Попробуйте с помощью функции из примера 3.8 вычислить F_{100}, F_{1000}, F_{10000}.

У (3.36) Написать функцию, вычисляющую число H_n (определено ниже) аналогично тому, как вычисляется F_n в примере 3.8.

$$H_n = n, \quad n < 3,$$
$$H_n = H_{n-1} + H_{n-2} + H_{n-3}, \quad n \geqslant 3.$$

Глава 4

Массивы и циклы

4.1. Циклы

Пример. Вычисление числа Фибоначчи F_n в примере 3.8 можно выразить как цикл.

```
double fib(int n) {
  double prev1 = 1.0, prev2 = 0.0;
  while (n--- > 0) {
    const double sum = prev1 + prev2;
    prev2 = prev1;
    prev1 = sum;
  }
  return prev2;
}
```

Пример. Считать n, вычислить двойной факториал $(n)!!$

$$0!! = 1,$$
$$(2k)!! = 2 \cdot 4 \cdot 6 \cdots 2k,$$
$$(2k+1)!! = 1 \cdot 3 \cdot 5 \cdots (2k+1).$$

```
// двойной факториал n
double dfact(int n) {
```

```
  double prod = 1.0;
  for (; n > 0; n -= 2)
    prod *= double(n);
  return prod;
}
```

Пример. Напечатать степени двойки вплоть до 2^{31}.

```
int main() {
  for (unsigned degree = 0, power = 1;
    degree < 32u; // условие — показатель < 32
    ++degree, power *= 2)
    cout << degree << ":\t" << power << '\n';
}
```

Фрагмент `++degree, power *= 2` в секции инкремента цикла `for` использует операцию `,` (запятая), которая вычисляет сначала левый операнд, затем правый операнд и возвращает его же (значение левого операнда отбрасывается).

Пример. Написать функцию, для заданных x и n вычисляющую $\sum_{i=1}^{n} \frac{x^i}{i!}$.

```
double computeSum(double x, int n) {
  double xi = x, ifact = 1.0, sum = x;
  for (int i = 2; i < n; ++i) {
    xi *= x;       // x в степени i
    ifact *= i;    // факториал i
    sum += xi / ifact; // сумма
  }
  return sum;
}
```

Пример. Вычислить x^n, где $n \in \mathbb{N}$. Вместо того чтобы умножать x на себя n раз, можно воспользоваться ассоциативностью произведения и двоичным представлением n, чтобы вычислить x^n, выполнив лишь $O(\log n)$ умножений. Обозначим i-й бит двоичного представления числа n через n_i. Тогда:

$$x^n = x^{n_0} \cdot (x^2)^{n_1} \cdot (x^4)^{n_2} \cdots (x^{2^i})^{n_i} \cdots .$$

Пример 4.1. Возведение в натуральную степень

```
double power(double x, unsigned n) {
  double result = 1.0;
  for (; n != 0; n /= 2) {
    if (n & 1) // домножить на текущий "квадрат"
      result *= x;
    x *= x; // следующий "квадрат"
  }
  return result;
}
```

У (4.1) Доработать пример 4.1, чтобы вычислять так же степени с отрицательными показателями (`int n`).

У (4.2) Напечатать таблицу умножения. Использовать двойной цикл `for`.

Далее предполагается, что последовательности чисел вводятся с клавиатуры. Программа должна заканчивать ввод при вводе признака конца файла.

У (4.3) Найти максимум и минимум последовательности.

У (4.4) Найти первый отрицательный член последовательности $a_n = \sin(\operatorname{tg} n), n \in \mathbb{N}$, вывести соответствующие n и a_n.

У (4.5) Посчитать количество смен знака в последовательности чисел, не содержащей нулей.

У (4.6) Считать n, вычислить $\cos(1 + \cos(2 + \ldots + \cos(n - 1 + \cos n))\ldots))$.

У (4.7) Посчитать две суммы: всех отрицательных и всех положительных чисел в последовательности.

У (4.8) Считать n, вычислить произведение $\prod\limits_{k=2}^{n}(1 - k^{-2})$.

У (4.9) Проверить, является ли последовательность чисел монотонной.

У (4.10) Считать n, вычислить n-й член последовательности

$$x_0 = 1, \quad x_n = nx_{n-1} + \frac{1}{n}.$$

У (4.11) Считать n и x, вычислить (не используя функцию `pow`) $\sum_{i=1}^{n} \cos x^i$.

У (4.12) Вычислить арифметическое среднее am последовательности чисел a_i.

У (4.13) Вычислить геометрическое среднее gm последовательности чисел a_i:

$$\mathrm{am}(\{a_i\}_{i=1}^n) = \frac{1}{n}\sum_{i=1}^{n} a_i, \quad \mathrm{gm}(\{a_i\}_{i=1}^n) = \left(\prod_{i=1}^{n} a_i\right)^{\frac{1}{n}}.$$

У (4.14) Вывести номера всех «счастливых» билетов, используя простой перебор (шестиуровневый цикл). Номер билета шестизначный и имеет вид «*abc def*», где $a, b, \ldots, f$ — десятичные цифры. Билет считается «счастливым», если верно равенство $a + b + c = d + e + f$.

Пример. Требуется найти некоторый корень уравнения $f(x) = 0$, где $f(x)$ — непрерывная функция на отрезке $[a, b]$, и известно, что $f(a)f(b) < 0$ (таким образом, хотя бы один корень на $[a, b]$ существует). Одним из простейших способов сделать это (приближенно, с абсолютной точностью `eps`) — использовать метод деления отрезка пополам. Чтобы сделать реализацию метода относительно общей, можно передавать ей функцию $f(x)$ как указатель на функцию.

```
double div2root(double (*f)(double),
      double a, double b, double eps) {
  while (eps < b - a) {
    const double mid = 0.5*(a + b), fm = f(mid);
    if (fm == 0.0) return mid;
    if (f(a) * fm < 0.0) b = mid;
    else                 a = mid;
  }
  return 0.5*(a + b);
}
```

У (4.15) Реализовать *метод секущих* поиска корня уравнения $f(x) = 0$. В методе секущих следующее приближение x_n к корню вычисляется через два предыдущих x_{n-1} и x_{n-2} как точка пересечения оси абсцисс прямой, проведенной через точки $(x_{n-1}, f(x_{n-1}))$ и $(x_{n-2}, f(x_{n-2}))$:

$$x_n = x_{n-1} - f(x_{n-1})\frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})}.$$

В качестве первых двух приближений можно взять границы отрезка $[a, b]$, использовавшегося выше в методе деления отрезка пополам. Сравнить количество итераций, выполняемых методом секущих и методом деления отрезка пополам, на ряде примеров.

4.2. Массивы

Предположим, что требуется посчитать скалярное произведение двух векторов $a = (a_0, a_1, a_2)^\top$ и $b = (b_0, b_1, b_2)^\top$.

```
float a0, a1, a2; // вектор a
float b0, b1, b2; // вектор b
cin >> a0 >> a1 >> a2; // считали a
cin >> b0 >> b1 >> b2; // считали b
float a_dot_b = a0 * b0 + a1 * b1 + a2 * b2;
cout << a_dot_b; // скалярное произведение
```

Очевидно, что такой подход к работе с векторами очень быстро исчерпывает себя (представьте стомерные вектора). Поэтому требуется обобщение: обращение к компонентам вектора по индексу, задаваемому переменной (что позволит записывать циклы), а не индексу, «зашитому» в код в виде части названия переменной. Таким обобщением является **массив** (*array*): набор значений одного типа, размещенных в памяти непосредственно друг за другом, к которым можно обращаться по имени массива и индексу элемента. Перепишем пример со скалярным произведением, заменив переменные-компоненты на массивы (пока без цикла).

```
float a[3]; // вектор a — 3 элемента
float b[3]; // вектор b — 3 элемента
cin >> a[0] >> a[1] >> a[2]; // считали a
cin >> b[0] >> b[1] >> b[2]; // считали b
float a_dot_b = a[0] * b[0] + a[1] * b[1]
              + a[2] * b[2];
cout << a_dot_b; // вывели произведение
```

Так как в качестве индекса массива может выступать значение выражения, вычисляемое во время выполнения программы, мы можем заменить константы 0, 1, 2 на переменную-счетчик цикла (здесь подходит `for`, поскольку имеет место перебор «от — до»).

```
float a[3]; // вектор a — 3 элемента
float b[3]; // вектор b — 3 элемента
for (int i = 0; i < 3; ++i)
  cin >> a[i]; // считали a
for (int i = 0; i < 3; ++i)
  cin >> b[i]; // считали b
float a_dot_b = 0.0;
for (int i = 0; i < 3; ++i)
  a_dot_b += a[i] * b[i];
cout << a_dot_b; // вывели произведение
```

У (4.16) Переписать пример «скалярное произведение», вынеся код, считывающий вектор, и код, вычисляющий скалярное произведение, в отдельные функции. Размерность векторов должна задаваться глобальной константой.

В следующей группе заданий предполагается, что используется заранее заданный массив фиксированного размера (статический массив). Возможно заполнение массива чтением элементов из `cin`, однако в примере ниже значения просто «зашиты» в программу.

Пример. Считать x, определить, сколько чисел из массива превосходят x. Напишем функцию, принимающую произвольный массив, а проверку выполним для конкретного массива.

```
// массив по адресу a длины size
int countGreater(float a[], int size, float x) {
  int result = 0;
  for (int i = 0; i < size; ++i) {
    if (x < a[i])
      ++result;
  }
  return result;
}

int main() {
  // массив с заранее заданными
  // значениями для проверки
  float a[7] = { 1, 2.4, 5.0, -0.33,
                 -0.1, 1e+30, 2e-30 };
  // запросим x
  cout << "x = ";
  float x = 0.0;
  cin >> x;
  // проверим функцию countGreater
  cout << countGreater(a, 7, x);
}
```

У (4.17) Вычислить геометрическое среднее (см. У 4.13) элементов некоторого массива.

У (4.18) Найти максимум и минимум среди элементов некоторого массива.

У (4.19) Вычислить скалярное произведение двух n-мерных векторов, заданных массивами.

У (4.20) Посчитать количество смен знака в последовательности чисел, заданной массивом. Для простоты можно считать, что массив не содержит нулей.

У (4.21) Посчитать количество цифр в строке.

У (4.22) Проверить, является ли массив монотонной последовательностью.

У (4.23) Заполнить массив длины n значениями x_i, где

$$x_0 = 1, \quad x_n = nx_{n-1} + \frac{1}{n}.$$

У (4.24) Вычислить арифметическое среднее (см. У 4.12) элементов некоторого массива.

У (4.25) Вычислить евклидову норму n-мерного вектора, заданного массивом:

$$\|x\| = \left(\sum_{i=1}^{n} x_i^2 \right)^{\frac{1}{2}}.$$

У (4.26) Завести два массива: x-координаты и y-координаты точек. Найти точку, наиболее удаленную от нуля.

Пример. С переменной типа `std::string` можно работать как с массивом символов. Следующая функция выводит переданную ей строку задом-наперед.

```
void printReversed(string s) {
  for (int i = s.size() - 1; i > -1; --i)
    cout << s[i];
}
```

Пример. Реализовать циклический сдвиг строки на заданное число символов. При циклическом сдвиге длина не изменяется, а «выпавшие» символы вставляются с противоположного конца строки. Например, если сдвинуть строку «alpha» циклически на 3 символа влево, получим строку «haalp».

```
// циклический сдвиг строки на n символов
// будем считать положительное n сдвигом влево,
//   а отрицательное n — сдвигом вправо
void rotate(string *s, int n) {
  string ts = *s; // временная копия строки
  // пройдем по всем символам исходной строки
  // и скопируем символ из копии туда,
  // где он должен стоять после сдвига
```

```
  const int len = s->size(); // длина строки
  n = n % len + len; // теперь 0 <= n < 2 * len
  for (int i = 0; i < len; ++i)
    (*s)[i] = ts[ (i + n) % len ];
  // для понимания этой формулы рекомендуется
  // промоделировать операцию на бумаге
}
```

У (4.27) Реализовать циклический сдвиг строки более эффективно, чем в примере: не используя временную копию и вычисление остатка от деления в цикле (можно один раз вычислить остаток перед циклом). Количество выполняемых операций не должно существенно зависеть от того, на сколько символов сдвигается строка.

Пример. Дан массив чисел. Дано число s, которое будем называть «целевым значением суммы». Найти внутри массива подпоследовательность чисел, сумма которой равна s.

```
/* Возвращает индекс первого элемента найденной
   последовательности или -1,
   если найти не удалось,
 a — адрес массива;  n — длина массива;
 sum — целевое значение суммы;
 length — адрес, по которому записать длину
       найденной последовательности. */
int parsum(int *a, int n,
           int sum, int *length) {
  // перебираем индекс первого элемента
  for (int i = 0; i < n; ++i) {
    int s = 0;
    // суммируем элементы, начиная с a[i]
    for (int j = i; j < n; ++j) {
      s += a[j];
      if (s == sum) { // нашли
        if (length) // ненулевой указатель?
          *length = 1 + j - i; // длина
        return i; // стартовый индекс
```

```
      }
    }
  }
  return -1; // не нашли
}
```

Пример. Реализовать «решето Сундарама». Из последовательности N нечетных натуральных чисел исключить все числа, индексы которых представимы в виде $i+j+2ij$, где $i, j \in \mathbb{N}$. Останутся только простые числа.

Пусть есть нечетное число, равное $2k+1$, тогда его индекс в последовательности равен k. Если число $2k+1$ составное, то выполняется равенство $2k+1 = (2i+1)(2j+1)$, откуда $k = i+j+2ij$. Перебирая различные натуральные i и j, такие, что $i+j+2ij < N$, фиксируем полученные числа в булевском массиве, используя k в качестве индекса.

```
// считаем, что массив sieve заполнен false
void sundaram(bool *sieve, int n) {
  for (int i = 1; 3 * i + 1 < n; ++i)
    for (int j = 1; j <= i; ++j) {
      const int k = i + j + 2 * i * j;
      if (n <= k) break;
      sieve[k] = true;
    }
}
// количество исходных нечетных чисел
const int N = 100;
int main() {
  bool sieve[N] = {false}; // заполним false
  sundaram(sieve, N);
  // выведем результат
  for (int i = 1; i < N; ++i) if (!sieve[i])
      cout << (2 * i + 1) << endl;
}
```

У (4.28) Написать и проверить две функции.

Функция `mismatchForward`. Даны две строки, найти позицию первого несовпадения этих двух строк. *Пример.* Даны строки «alpha» и «alps», ответом будет число 3 (‘h’≠‘s’).

Функция `mismatchBackward`. Даны две строки, найти позицию (отсчитывая с начала первой строки) первого с конца несовпадения этих двух строк. *Пример.* Даны строки «magister» и «master», ответом будет 3 (‘i’≠‘a’).

У (4.29) Даны две строки. Написать и проверить функцию, которая ищет вторую строку в первой, возвращая -1, если вторая строка не найдена в первой (не является ее подстрокой), иначе возвращая (первую, считая с начала первой строки) позицию второй строки в первой строке. *Пример.* Даны строки «base after» и «aft», ответом будет 5.

У (4.30) Провести N чисел через «решето Эратосфена».

Каждое число первоначально «не выколото», удаляемые числа «выкалываются». Таким образом, можно завести массив булевских значений размера N, каждый элемент которого хранит `true`, если число, равное индексу этого элемента, выколото, и `false` в обратном случае. Алгоритм можно представить в виде двойного цикла.

Внешний цикл. Начиная с индекса 2, пробегаем массив, пока он не кончится. Если в процессе встретили `false`, то выполняем внутренний цикл. Если массив кончился, выходим из цикла и выводим индексы элементов, равных `false` (результат работы алгоритма).

Внутренний цикл. Присваиваем `true` всем последующим элементам массива с шагом, равным индексу найденного элемента.

У (4.31) Заполнить массив длины N простыми числами методом перебора с проверкой взятием остатка от деления.

Первому элементу массива присваиваем значение 2. Перебирая $i \geqslant 3$ до тех пор, пока не заполним массив, проверяем, делится ли i на ранее найденные простые числа (которые уже хранятся в массиве); если не делится, то дописываем его в следующую позицию в массиве.

Для проверки i на простоту достаточно проверять остаток от деления i на простые числа в диапазоне $[2, \lfloor\sqrt{i}\rfloor]$.

У (4.32) Дано целое число p и база r системы счисления, $2 \leqslant r \leqslant 36$. Сформировать строку представления числа p в позиционной системе счисления с базой r. «Недостающие» цифры (сверх 9) в системах счисления с $r > 10$ обозначаются латинскими буквами от 'a' до 'z'. Например, в 36-ричной системе для записи чисел используются цифры '0', '1', ..., '9', 'a', 'b', ..., 'y', 'z'. Алгоритм оформить как функцию вида

```
string itos(int p, int r) {
  char digits[66] = {}; // забить нулями
  char *p = digits + 64;// позиция младшей цифры
  /* заполнить digits — собственно алгоритм */
  return string(p); // создать string из массива
}
```

Строку `string` можно сформировать как копию локального массива `char`, обязательно завершающегося символом с кодом `0`. Для записи r-ичного представления `int` должно хватить 66 знаков (64 цифры для двоичного 64-битного целого, место для знака, место для завершающего нуля).

У (4.33) Дана строка и целое число $2 \leqslant r \leqslant 36$. Распознать строку как запись целого числа в r-ичной системе счисления, получить это число (например, в виде значения типа `int`). Алгоритм оформить как функцию

```
int stoi(const string &s, int r);
```

У (4.34) Распечатать последовательность чисел Фибоначчи в двоичной системе счисления, выравнивая числа по младшему разряду (можно выводить разряды слева направо).

У (4.35) Посчитать количество смен знака прироста в последовательности чисел, введенной с клавиатуры. В последовательности может встретиться произвольное количество равных чисел, идущих подряд.

С-строкой называется массив символов, последний из которых имеет числовое значение 0 («завершающий нуль») и слу-

жит признаком конца строки. Благодаря завершающему нулю строкой можно оперировать через указатель на массив (например, типа `char*`), не храня ее длину.

Написать функции, работающие со строками через указатели на `char`.

У (4.36) `cslen`: подсчет длины строки;

У (4.37) `cseq`: сравнение двух строк на равенство;

У (4.38) `cscopy`: копирование одной строки в другую (предполагая, что массив, куда происходит копирование, имеет достаточно большой размер);

У (4.39) `cscat`: конкатенацию двух строк (дописать вторую строку в конец первой, считая, что объем выделенной на нее памяти позволяет это);

У (4.40) `csucopy`: скопировать коды символов строки (кроме завершающего нуля) в массив `unsigned`, считая, что он передается функции по указателю и имеет достаточно большой размер, чтобы принять строку;

У (4.41) `cstoupper`: преобразовать все маленькие буквы в большие (использовать стандартную функцию `toupper` из заголовочного файла `<cctype>`);

У (4.42) `cstolower`: преобразовать все большие буквы в маленькие (использовать стандартную функцию `tolower` из заголовочного файла `<cctype>`).

Пример. Поиск символа в С-строке. Если символ не найден, возвращает нулевой указатель.

```
char* cschr(char *str, char chr)
{ while (*str && *str != chr) ++str;
  return *str? str: nullptr; }

int main()
{ char *s = "omg much more";
  cout << (cschr(s, 'e') - s); }
```

У (4.43) Задание состоит из трех частей.

1. Посчитать частоты символов в строке, вывести ненулевые частоты (и соответствующие им символы и их коды) на экран. Под *частотой* символа с кодом k в строке понимается количество символов с кодом k в этой строке, поделенное на длину строки. Частота символа — число с плавающей точкой в диапазоне $[0, 1]$.

Для этого организовать массив из 256 целых. Проходя строку посимвольно циклом, увеличивать на единицу элемент массива с индексом, равным коду текущего символа (предварительно приведенному к `unsigned char`). Подсчет символов по кодам выполнять в отдельной функции:

```
void countCodes(const std::string &text,
                int amount[256]) {
  // обнулить amount
  // пройти по text,
  //   увеличивая соответствующие элементы amount
}
```

Вывод ненулевых частот осуществлять с помощью функции

```
void printFrequencies(int amount[256],
                      int textLength) {
  // пройти по amount,
  //   для ненулевых элементов выводя:
  //     номер кода, символ, частоту
}
```

2. Считать две строки: «текст» и «ключ». Произвести шифрование с использованием поразрядной операции *исключающее или* (будем называть этот алгоритм «XOR-гаммирование», он описан ниже, в комментарии внутри функции `xorGamma`). Вывести результат (зашифрованный текст) посимвольно.

Благодаря свойствам операции *исключающее или* повторное выполнение XOR-гаммирования с зашифрованным текстом и тем же ключом производит исходный текст.

Выполнить повторное XOR-гаммирование зашифрованного текста и вывести полученную строку (чтобы видеть, что она действительно совпадает с исходным текстом). XOR-гаммирование реализовать в виде отдельной функции

```
void xorGamma(std::string &text,
        const std::string &key) {
  // для всех i от 0 до (text->size() - 1):
  //    i-й символ text ^=
  //          (i % key.size())-й символ key
}
```

Вопрос 1. Пусть длина ключа произвольна и заранее неизвестна. Как реализовать алгоритм без применения операции взятия остатка от деления?

Вопрос 2. Пусть длина ключа есть фиксированная степень двойки. Чем следует заменить операцию взятия остатка для ускорения работы алгоритма?

Вопрос 3. Пусть пароль пользователя является произвольной строкой. Каким способом из него можно получить ключ, длина которого есть степень двойки? (Желательно, чтобы ключ был похож на случайный набор символов.)

3. Объединить первые две части. Считать текст и ключ. Вывести частоты символов в исходном тексте и в зашифрованном тексте. Проанализировать результат для текста на естественном языке и короткого ключа (например, сло́ва на том же языке).

Вопрос 4. Если длина ключа значительно меньше длины текста, то может оказаться эффективной атака, сводящаяся к анализу распределения частот символов, расположенных друг от друга на расстоянии длины ключа. Как с этим можно бороться? Пусть длина исходного ключа задана извне.

4.3. Перечисления

Пусть день недели задается числом от 1 до 7 (номер дня недели). Напишем функцию, сообщающую английское название дня недели по его номеру.

```
string nameOfDay(int number) {
  if (number == 1) return "Sunday";
  if (number == 2) return "Monday";
  if (number == 3) return "Tuesday";
  if (number == 4) return "Wednesday";
  if (number == 5) return "Thursday";
  if (number == 6) return "Friday";
  if (number == 7) return "Saturday";
  return "Unknown"; // ни один из вариантов
}
```

Вместо того чтобы оперировать числовыми значениями номеров непосредственно, можно ввести (глобальные) константы. Использование имен вместо чисел упрощает восприятие текста программы.

```
const int Sunday    = 1, Monday    = 2,
          Tuesday   = 3, Wednesday = 4,
          Thursday  = 5, Friday    = 6,
          Saturday  = 7;
```

Теперь в коде функции `nameOfDay` можно заменить числа на имена констант.

C++ предлагает более удобный способ введения последовательностей целочисленных констант, чем показанный выше. Для этого следует воспользоваться **перечислением**. Перечисление представляет собой тип, множество значений которого задано набором констант. Для определения перечисления и сопутствующих констант используется слово `enum`.

```
enum DayOfWeek {
  Sunday = 1, Monday, Tuesday, Wednesday,
  Thursday, Friday, Saturday };
```

Теперь `DayOfWeek` — тип, переменные которого могут принимать значения `Sunday`, `Monday` ... `Saturday`. Впрочем, за ними скрываются целочисленный тип и целые числа, поэтому доступны преобразования из целых чисел (явно) и в целые числа (неявно). Явно задавать значения констант не обязательно, пропущенные значения компилятор сгенерирует сам, прибавив единицу к значению предыдущей константы.

В случае перебора целочисленных значений каскад `if-else` лучше заменить на `switch-case`.

```
string nameOfDay(DayOfWeek number) {
  switch (number) {
  case Sunday:    return "Sunday";
  case Monday:    return "Monday";
  case Tuesday:   return "Tuesday";
  case Wednesday: return "Wednesday";
  case Thursday:  return "Thursday";
  case Friday:    return "Friday";
  case Saturday:  return "Saturday";
  default:        return "Unknown";
  }
}
```

Так как значения номеров дней недели идут подряд, вместо `switch-case` можно использовать обращение к массиву.

```
string nameOfDay(DayOfWeek number) {
  static const char* dayName[] = {
        "Sunday", "Monday", "Tuesday",
        "Wednesday", "Thursday", "Friday",
        "Saturday" };
  if (Sunday <= number && number <= Saturday)
    return dayName[(int)number - 1];
  return "Unknown";
}
```

4.4. Конечные автоматы

Конечным автоматом будем называть машину, которая:

- работает в пошаговом режиме;
- начав с некоторого (начального) состояния, на каждом шаге считывает символ из входного потока;
- переходит в новое состояние в зависимости от текущего состояния и считанного символа, следуя заранее определенному конечному набору правил;
- может иметь финальное состояние, после попадания в которое работа автомата считается законченной;

и при этом в каждый момент времени находится в одном из конечного числа заранее определенных состояний.

Конечные автоматы удобно изображать в виде диаграмм, где показаны состояния, связанные стрелками, обозначающими переходы при считывании того или иного символа. Например, на рис. 4.1 изображен автомат, распознающий строковый литерал языка C++ (упрощенный).

В качестве примера рассмотрим конечный автомат, решающий задачу нормализации переводов строк. Среди управляющих символов кодировки ASCII есть два символа, традиционно используемых для оформления переводов строк: код 10 (`LF`, сдвинуть каретку на строку вниз) и код 13 (`CR`, вернуть каретку в начало строки). Есть четыре основных варианта: `LF` (Unix-подобные системы), `CR LF` (Windows, сетевые протоколы), а также `CR` и `LF CR`, не используемые в современных системах. Схема автомата показана на рис. 4.2.

Итак, наш конечный автомат будет заменять во входном тексте все четыре варианта переводов на `'\n'`, а прочие символы оставлять неизменными. Вывод перевода строки обозначен `> \n`, вывод последнего считанного символа обозначен `> *`, для уменьшения загруженности схемы условие «прочитан символ,

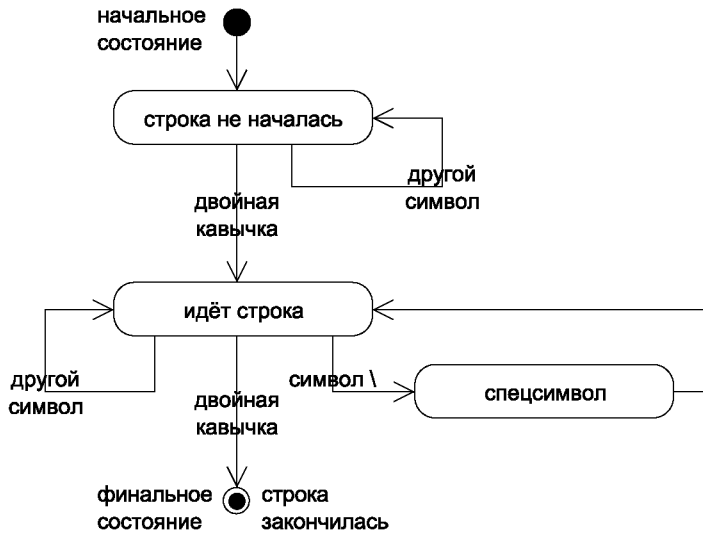

Рис. 4.1. Автомат для распознавания строковых литералов

отличный от CR и LF» опущено. Выход происходит, когда входные данные прочитаны до конца.

Конечный автомат удобно реализовывать в виде перечисления состояний и цикла с вложенным `switch`, выполняющим действия в соответствии с текущим значением переменной, хранящей состояние (ниже `state`). Входные данные будем получать как диапазон в массиве `char`, результат писать в массив `char`, переданный указателем на начальный элемент. Функция, реализующая автомат, возвращает указатель на символ, следующий за последним записанным символом.

Пример 4.2. Автомат на основе switch

```
char* normalize
(const char *from, const char *end, char *to) {
  enum { Other, LF, CR } state = Other;
  while (from != end) {
    const char ch = *from++;
```

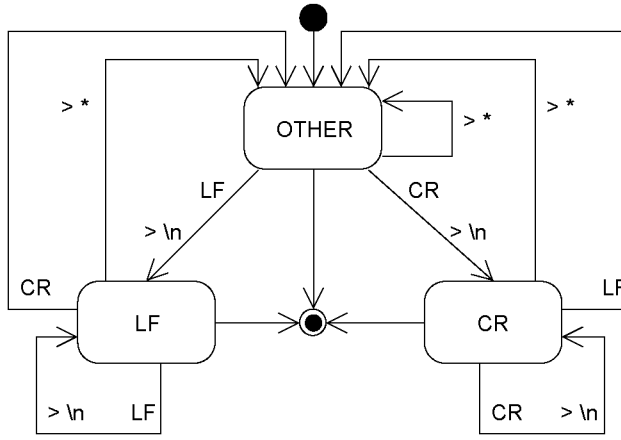

Рис. 4.2. Автомат «CRLF»

```
switch (state) {
case Other:
  switch (ch) {
  case '\n':
    *to++ = '\n'; state = LF; break;
  case '\r':
    *to++ = '\n'; state = CR; break;
  default:
    *to++ = ch; }
  break;
case LF:
  switch (ch) {
  case '\n':
    *to++ = '\n'; break;
  case '\r':
    state = Other; break;
  default:
```

```
        *to++ = ch; state = Other; }
      break;
    case CR:
      switch (ch) {
      case '\n':
        state = Other; break;
      case '\r':
        *to++ = '\n'; break;
      default:
        *to++ = ch; state = Other; }
      break;
    }
  }
  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`. Его преимущество перед использованием `switch` — в потенциально большей производительности полученного машинного кода (роль переменной, хранящей текущее состояние автомата играет регистр процессора, называемый *указатель инструкции*) и иногда в краткости. Его недостаток — меньшая гибкость.

```
char* normalize
(const char *from, const char *end, char *to) {
  char ch;
Other:
  if (from == end) return to;
  ch = *from++;
  if (ch == '\n') goto LF;
  if (ch == '\r') goto CR;
  *to++ = ch;
  goto Other;
LF:
```

```
    *to++ = '\n';
    if (from == end) return to;
    ch = *from++;
    if (ch == '\r') goto Other;
    if (ch == '\n') goto LF;
    *to++ = ch;
    goto Other;
CR:
    *to++ = '\n';
    if (from == end) return to;
    ch = *from++;
    if (ch == '\n') goto Other;
    if (ch == '\r') goto CR;
    *to++ = ch;
    goto Other;
}
```

Оба варианта реализации автомата (и с использованием `switch`, и с использованием `goto`) представляют собой простую интерпретацию схемы автомата на языке программирования. Подобные действия могут быть автоматизированы.

Анализ кода с `goto` позволяет переписать «явную» передачу управления в виде циклов, убрав метки и инструкции `goto`, придав ему, таким образом, структурный вид без необходимости вводить переменную состояния.

```
char* normalize
(const char *from, const char *end, char *to) {
  char ch;
  while (from != end) {
    switch (ch = *from++) {
    case '\n':
      do {
        *to++ = '\n';
        if (from == end) return to;
        ch = *from++;
      } while (ch == '\n');
```

```
        if (ch != '\r') *to++ = ch;
        break;
      case '\r':
        do {
          *to++ = '\n';
          if (from == end) return to;
          ch = *from++;
        } while (ch == '\r');
        if (ch != '\n') *to++ = ch;
        break;
    default:
      *to++ = ch;
    }
    return to;
}
```

У (4.44) Реализовать конечный автомат с рис. 4.1.

У (4.45) Определить количество предложений в тексте, считая, что каждое предложение заканчивается по крайней мере одной буквой, за которой следует последовательность точек, восклицательных или вопросительных знаков, а затем или заканчивается ввод (например, конец файла), или идет последовательность пробельных символов и заглавная буква. Построить конечный автомат (рис. 4.3).

У (4.46) «Исправить» маленькие буквы, начинающие предложения (см. предыдущее задание).

У (4.47) Проверить, правильно ли расставлены круглые скобки в тексте.

Примеры. Правильно: «abc», «1+()−2», «(()())(x)».

Неправильно: «)m(», «())», «ome(ga».

Замечание. Строго говоря, конечный автомат не может распознать произвольную скобочную форму, так как для этого требуется бесконечное число состояний (каждое состояние отвечает своей глубине вложенности), однако здесь можно ограничиться некоторой максимальной глубиной.

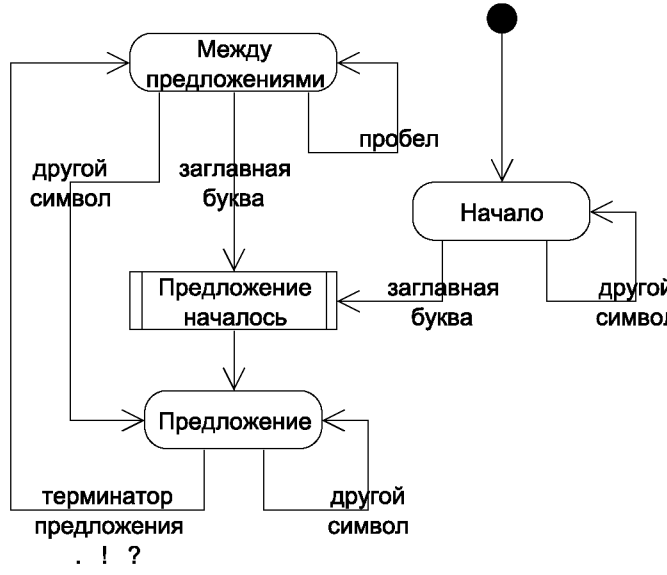

Рис. 4.3. Автомат для подсчета числа предложений

У (4.48) Дана строка `code`. Сформировать на ее основе новую строку, содержащую символы строки `code` в исходном порядке, но без подряд идущих повторений, вместо этого указывая количество подряд идущих повторяющихся символов.

Пример. «aaaAAAAbbbc» ⇒ «a3A4b3c1».

Замечание. Справедливо замечание, аналогичное замечанию в предыдущем упражнении: для конечности достаточно задать максимально возможное количество повторений одного символа. Если символ продолжает повторяться, его можно считать за другой символ.

У (4.49) Составить схему конечного автомата, отвечающего простому калькулятору, вычисляющему арифметические выражения в инфиксной форме без учета приоритета операций. Составить программу, реализующую данный конечный автомат. Калькулятор должен позволять вводить числа в формате с плавающей точкой и поддерживать не менее пяти арифметических действий: сложение, вычитание, умножение, деление, взятие остатка от деления, возведение в степень.

У (4.50) Для произвольного числа x вывести все возможные тождества вида $1 \circ 2 \circ 3 \circ 4 \circ 5 \circ 6 \circ 7 \circ 8 \circ 9 = x$, где вместо $\circ$ должен стоять знак операции $+$, $-$, $\times$ или не должно быть ничего (т. е. цифры могут идти подряд, формируя запись числа из нескольких цифр).

Пример. Для $x = 2002$ программа должна вывести:

```
1 x 23 + 45 x 6 x 7 + 89
1 x 2 + 34 x 56 + 7 + 89
```

Глава 5

Структуры данных и файлы

Реализовать стек и очередь (все базовые операции стека и очереди должны выполняться, используя количество действий, не зависящее от количества хранимых элементов) на основе:

У (5.1) линейного односвязного списка, при этом:

- **стек** управляется через один указатель на вершину, каждый элемент стека хранит указатель на нижележащий элемент, у самого нижнего элемента (помещенного в стек первым и наиболее удаленного от вершины) этот указатель равен `nullptr`;
- **очередь** управляется через два указателя: на «начало» (где происходит вставка) и «конец» (где происходит удаление), при прохождении списка по указателям, хранимым в звеньях, движение происходит от «конца» в сторону «начала»;

У (5.2) кольцевого односвязного списка, при этом:

- **стек** управляется через указатель на первый (самый нижний) элемент; так как список кольцевой, указатель в первом элементе указывает на вершину стека, что позволяет

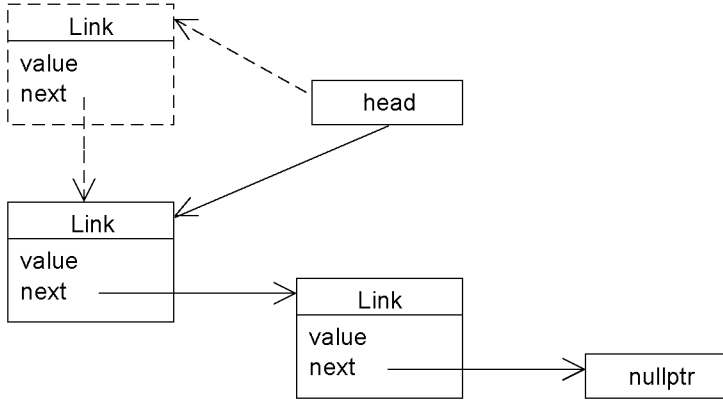

Рис. 5.1. Добавить звено в начало линейного односвязного списка – стека

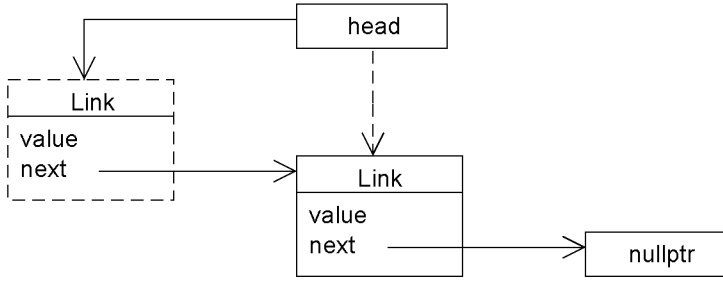

Рис. 5.2. Удалить звено с начала линейного односвязного списка – стека

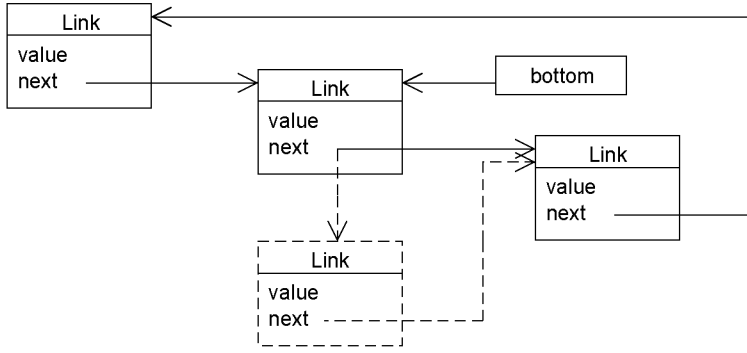

Рис. 5.3. Добавить звено в кольцевой односвязный список – стек

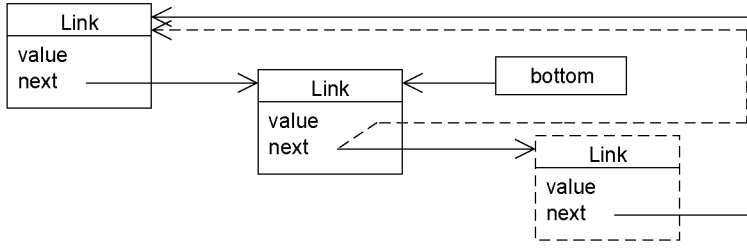

Рис. 5.4. Удалить звено из кольцевого односвязного списка – стека

как удалять, так и добавлять элементы в стек за постоянное число операций;

- **очередь** отличается от стека тем, что при вставке элемента управляющий указатель сдвигается на новый элемент, звено на которое он указывает, является последним, вставленным в очередь и хранит указатель на следующий на удаление элемент.

У (5.3) Используя реализованный стек, написать программу, проверяющую корректность расстановки скобок разного вида: `()` `[]` `{}`.

Корректно: «`()a[] - {}`», «`([+{beta}+])`», «`({ } [ ] )`».

Некорректно: «`)[]{}}`», «`(((]]]`», «`[} () {]`», «`([z)]`».

У (5.4) Дана строка `text`. Сформировать на ее основе новую строку, содержащую фрагменты `text`, не заключенные в круглые скобки (англ. *parentheses*).

Пример. «alpha(something)beta()((u)10)s» ⇒ «alphabetas».

Замечание. Незакрытые круглые скобки остаются, например: «((a)» ⇒ «(», «(a))» ⇒ «)».

```
// заголовок функции
string noParentheses(const string &text);
```

У (5.5) Дан односвязный список, возможно замкнутый сам на себя. Написать функцию, которая за линейное по размеру списка время и с использованием постоянного объема памяти определяет, замкнут этот список на себя или же является линейным (последнее звено имеет нулевой указатель на следующее звено). Функция не должна изменять исследуемый список.

У (5.6) Арифметическое выражение можно преобразовать в *постфиксную запись*, не опирающуюся на приоритеты операций и не требующую использования скобок. Пример постфиксной записи:

$$\frac{w-r}{z+t}(x+y) \quad \to \quad w \quad r \quad - \quad z \quad t \quad +/ \quad x \quad y \quad +*$$

При вычислении выражения, записанного в постфиксной записи, все считываемые числа помещаются в стек. При считывании знака операции ее операнды извлекаются из стека (два последних помещенных туда числа), результат вычисления помещается обратно в стек. На вершине стека всегда находится некоторый последний вычисленный результат (или последнее число):

$$\begin{array}{rrrrrrrrrrrrrrrrrrrrr}
 & & & & & 10 & 2 & - & 3 & 1 & +/ & 1.5 & 10 & +* & \to \\
8 & 3 & 1 & +/ & 1.5 & 10 & +* & \to & 8 & 4 & / & 1.5 & 10 & +* & \to \\
 & & & & & 2 & 1.5 & 10 & +* & \to & 2 & 11.5 & * & \to & 23.
\end{array}$$

Задание. Реализовать стек чисел. Реализовать калькулятор постфиксных выражений, выполняющий операции с числами с плавающей точкой (как минимум, сложение, вычитание, умножение, деление, вычисление остатка от деления, возведение в степень).

У (5.7) Реализовать автомат, вычисляющий значение переданного ему выражения в соответствии с грамматикой (должен поддерживать по крайней мере четыре базовых арифметических операции).

выражение $\to$ *число* | *S-форма*

S-форма $\to$ «(» *операция* {*выражение*} «)»

операция $\to$ «+» | «−» | «*» | «/»

Пример.

$$(+\ (*\ 2\ 3\ 4)\ (/\ 10\ 2)\ -1) \quad \to \quad (+\ 24\ 5\ -1) \quad \to \quad 28.$$

У (5.8) Пусть лабиринт задан двумерным массивом `bool`, индексы ячеек соответствуют их координатам. Ячейка содержит `true`, если она проходима, и `false`, если нет. За один ход двигаться разрешено либо на один шаг по вертикали, либо на один шаг по горизонтали. Построить автомат, находящий

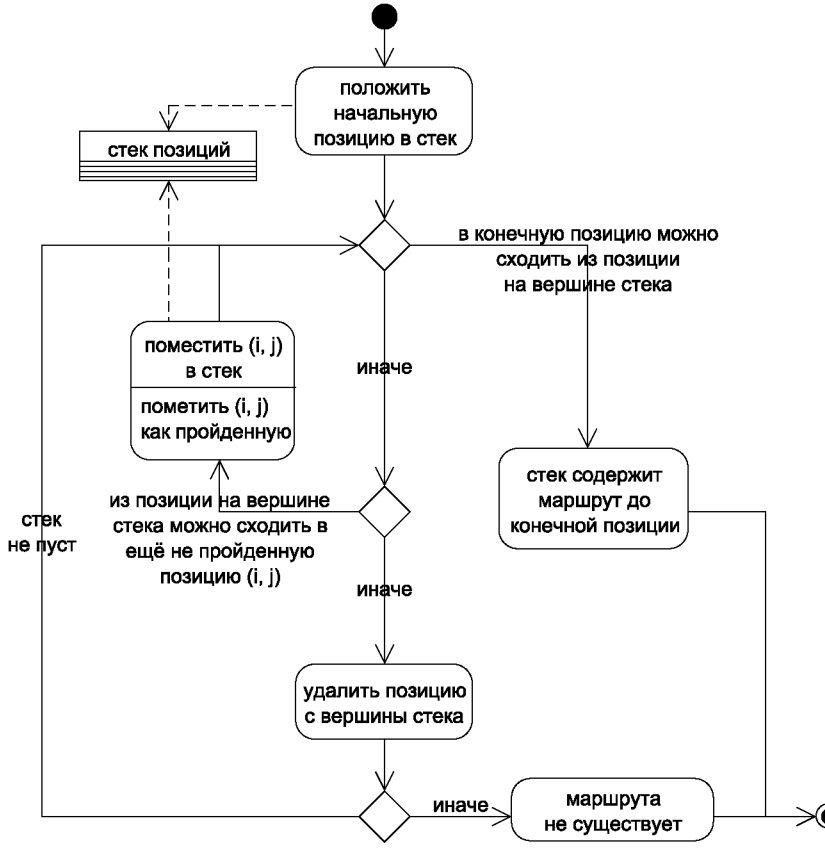

Рис. 5.5. Путь в лабиринте

маршрут между двумя заданными позициями (парами индексов), если он существует, либо сигнализирующий об отсутствии маршрута.

Схема автомата, решающего эту задачу, показана на рис. 5.5 (фактически это блок-схема алгоритма). Вместо того чтобы помечать позиции как пройденные, можно просто присваивать `false` соответствующей ячейке лабиринта (таким образом, уже нельзя будет сходить в эту ячейку повторно).

Пример. Длину файла средствами стандартной библиотеки C++ можно получить с помощью функций `seekg` и `tellg`.

```
streamsize file_size(const char *fn) {
  ifstream f(fn);
  return f? f.seekg(0, ios::end).tellg(): 0;
}
```

У (5.9) Реализовать дек на основе двусвязного списка, узлы которого хранят значения типа `std::string`. Написать проверочный код. Используя этот дек, написать программу, выполняющую следующие шаги.

1. Запросить у пользователя имя файла.
2. Открыть этот файл для чтения, если это возможно.
3. Считать содержимое файла построчно, сохраняя непустые строки в деке.
4. «Разобрать» его на несколько файлов так, чтобы строки каждой длины попадали в свой файл (в том же порядке, в котором они были размещены в исходном файле). Все строки должны попасть в какой-либо из новых файлов.

Например, если входной файл имеет имя input.txt и содержит строки длин 2, 3 и 4, то должно получиться три файла. Можно им дать названия 2.input.txt, 3.input.txt и 4.input.txt соответственно. Для получения имен файлов с номером можно использовать поток вывода в строку `std::ostringstream` из заголовочного файла `<sstream>`.

Замечание. Можно обойтись и без дека. Впрочем, манипуляции с деком позволяют не переоткрывать файлы при записи.

У (5.10) Дан текстовый файл, представляющий собой простейший словарь, где каждому слову и его переводу отведена своя строка. Требуется считать его и преобразовать в обратный словарь. Например:

```
второй second          fifth пятый
десятый tenth          first первый
первый first      ⇒    second второй
пятый fifth            tenth десятый
третий third           third третий
```

В словаре строки должны быть упорядочены по первому слову (лексикографически). Строки `std::string` можно сравнивать лексикографически, используя операцию `<`. Содержимое строк можно обменивать с помощью функции `std::swap`. Реализовать сортировку алгоритмом, работающим в среднем быстрее, чем за квадратичное время.

У (5.11) Составить программу, позволяющую просмотреть любую часть заданного файла в виде набора «строк» (по 16 байт в каждой строке). Каждая из строк начинается смещением от начала файла, за которым идут числовые значения содержащихся в ней байт (в шестнадцатеричной форме), и завершается символьным представлением этих байт в текущей кодировке. Необходимо подавлять символы перевода строки, табуляции и забоя, чтобы не разрывать строки (вместо них следует выводить пробел).

У (5.12) Реализовать арифметику над многочленами. Класс должен хранить упорядоченный по возрастанию степеней список пар (*показатель*, *коэффициент*). Например, многочлен

$$x^{12} + 8x^4 + 3x^2 - 12.5x - 1$$

соответствует списку

$$(0, -1), (1, -12.5), (2, 3), (4, 8), (12, 1).$$

Для класса чисел должны быть перегружены операторы `=`, `+`, `+=`, `-`, `-=`, `*`, `*=`, `/`, `/=`, `>>` (ввод из потока), `<<` (вывод в поток), а также операция вычисления значения многочлена в заданной точке (оператор `()`).

У (5.13) Числа можно представлять по-разному. Иногда возникает необходимость в полном соответствии арифметики «компьютерной» арифметике «человеческой». В таком случае обращаются к *десятичным числам*.

Десятичное число определяется как последовательность десятичных цифр. Десятичные цифры можно хранить по одной («неупакованное число») или по две («упакованное число») в байте. Если требуется дробная часть, то ее длину обычно фиксируют (например, не больше 20 разрядов), в то время как длина целой части не должна быть ограничена.

Реализовать операции с неупакованными десятичными числами с фиксированной запятой в три этапа.

1. Целое число фиксированной длины. Реализовать сложение, вычитание, умножение («столбиком») и деление, а также текстовый ввод и вывод чисел. Оформить числа в виде класса, содержимое представить в виде массива цифр (тип `char`) фиксированной длины (оформить константой) и знака. Для класса чисел должны быть перегружены операторы `=`, `+`, `+=`, `-`, `-=`, `*`, `*=`, `/`, `/=`, `>>` (ввод из потока), `<<` (вывод в поток), сравнения.

2. Целое число произвольной длины. Массив фиксированной длины заменить динамическим массивом, реализовать автоматическое увеличение длины при появлении новых старших разрядов. Определить операции копирования и деструктор.

3. Добавить дробную часть фиксированной длины. Модифицировать все функции класса «число» для корректной работы с дробной частью.

Приложение

Список рекомендуемой литературы

Ахо А. Структуры данных и алгоритмы / А. Ахо, Дж. Хопкрофт, Дж. Ульман. — М. : Вильямс, 2003. — 384 с.

Кормен Т. Алгоритмы : построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. — 3-е изд. — М. : Вильямс, 2013. — 1324 с.

Макконнелл Дж. Основы современных алгоритмов / Дж. Макконнелл. — 2-е изд. — М. : Техносфера, 2004. — 368 с.

Мэйерс С. Эффективное использование C++ / С. Мэйерс. — 3-е изд. — М. : ДМК Пресс, 2006. — 300 с.

Прата С. Язык программирования C++ : лекции и упражнения : учеб. / С. Прата. — 6-е изд. — М. : Вильямс, 2012. — 1244 с.

Страуструп Б. Программирование. Принципы и практика использования C++ / Б. Страуструп. — М. : Вильямс, 2011. — 1248 с.

Страуструп Б. Язык программирования C++ : спец. изд. / Б. Страуструп. — М. : Бином-Пресс, 2011. — 1136 с.

Стюарт Т. Теория вычислений для программистов / Т. Стюарт. — М. : ДМК Пресс, 2014. — 384 с.

Эккель Б. Философия C++. Введение в стандартный C++ / Б. Эккель. — 2-е изд. — СПб. : Питер, 2004. — 572 с.

Числа с плавающей запятой

Для работы с вещественными числами могут использоваться различные способы покрытия вещественной оси (конечным) набором представимых чисел. Как правило, они ограничиваются определенным подмножеством рациональных чисел:

- дроби вида m/n: пара целых чисел (m, n);
- числа с фиксированной запятой[1]: целое число m представляет дробь вида $m \cdot r^{-p}$, где r и p заданы заранее;
- числа с плавающей запятой: пара целых чисел (m, p) задает дробь вида $m \cdot r^p$, где r задано заранее. Таким образом, позиция запятой p 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. Наиболее распространенным стандартом представления и правил работы с числами с плавающей запятой является стандарт IEEE-754. Его поддерживают большинство современных процессоров, умеющих работать с числами с плавающей запятой «непосредственно».

IEEE-754 предполагает следующее двоичное представление числа.

знак s	**экспонента** E	**мантисса** M
1 разряд	e разрядов	m разрядов
старший бит	$(m+e-1)$ биты m	$(m-1)$ биты 0

[1]Вместо слова *запятая* часто используется слово *точка*, так как в англоязычных странах для разделения целой и дробной части числа традиционно используется точка.

Для *нормализованных* чисел в E хранится сдвинутое на $b = 2^{e-1} - 1$ значение показателя степени p (далее предполагается $r = 2$). Кроме «обычных» чисел, двоичное представление позволяет хранить специальные значения.

тип значения	E	M	значение
нуль	0	0	$(-1)^s \cdot 0$
денормализованное	0	$\neq 0$	$(-1)^s \cdot 0{,}(M) \cdot 2^{1-b}$
нормализованное	$[1, 2b]$	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

Замечания

- Денормализованные числа позволяют постепенно приблизиться к нулю (с постоянным шагом).

- Наличие двух нулей (отрицательного и положительного) иногда полезно: так, при получении очень малых величин мы по крайней мере сохраним знак (с какой стороны от нуля).

- Тем не менее оба нуля считаются равными друг другу. Кроме того, полагается $\sqrt{-0} = 0$, $\log(-0) = -\infty$, $0^0 = 1$.

- 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 неопределен).

- Для нечисел операторы сравнения всегда возвращают `false` (нечисло не равно само себе).

- Как правило, процессоры гораздо медленнее выполняют арифметические операции с денормализованными числами, бесконечностями и нечислами, чем с нормализованными числами, что может вызывать резкое падение производительности в некоторых случаях.

- Сумма и произведение чисел с плавающей запятой, строго говоря, не являются ассоциативными операциями.

Стандарт IEEE-754-2008 определяет четыре основных «двоичных» формата чисел с плавающей запятой.

формат	precision	бит	e	m	d	ε
binary16	half	16	5	10	5	$9{,}77 \cdot 10^{-4}$
binary32	single	32	8	23	9	$1{,}19 \cdot 10^{-7}$
binary64	double	64	11	52	17	$2{,}22 \cdot 10^{-16}$
binary128	quadruple	128	15	112	36	$1{,}93 \cdot 10^{-34}$

формат	мин. денорм.	мин. норм.	макс. норм
binary16	$5{,}96 \cdot 10^{-8}$	$6{,}1 \cdot 10^{-5}$	65504
binary32	$1{,}4 \cdot 10^{-45}$	$1{,}18 \cdot 10^{-38}$	$3{,}4 \cdot 10^{38}$
binary64	$4{,}94 \cdot 10^{-324}$	$2{,}2 \cdot 10^{-308}$	$1{,}8 \cdot 10^{308}$
binary128	$6{,}48 \cdot 10^{-4966}$	$3{,}4 \cdot 10^{-4932}$	$1{,}2 \cdot 10^{4932}$

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 десятичные дроби являются бесконечными в двоичной системе: например, 0.2 не представимо точно в указанных форматах. Двоичные же дроби хотя и представимы точно конечным числом знаков, могут потребовать сотни десятичных цифр для точной записи.

Язык C++ предоставляет три встроенных типа для работы с числами с плавающей точкой:

- `float` обычно соответствует IEEE-754 binary32;
- `double` обычно соответствует IEEE-754 binary64;
- `long double` на x86 нередко отождествляется с 80-битным форматом сопроцессора Intel 8087, в то время как Visual C++ использует представление, идентичное `double`, а некоторые другие компиляторы могут использовать IEEE-754 binary128.

Часть характеристик форматов с плавающей запятой, используемых конкретным компилятором, можно получить, подключив `<cfloat>`, где определен ряд макросов. Ниже `*` соответствует `FLT` для типа `float`, `DBL` для `double` и `LDBL` для `long double`:

- `FLT_RADIX` база системы счисления r (обычно 2);
- `*_MANT_DIG` разряды полной мантиссы $(m + 1)$;
- `*_DIG` достаточное число десятичных знаков после запятой d;
- `*_MIN_EXP` минимальная смещенная экспонента для нормализованных;
- `*_MIN_10_EXP` минимальная десятичная экспонента для нормализованных;
- `*_MAX_EXP` максимальная смещенная экспонента для нормализованных;
- `*_MAX_10_EXP` максимальная десятичная экспонента;
- `*_MIN` наименьшее положительное нормализованное число;
- `*_MAX` наибольшее конечное представимое число;
- `*_EPSILON` наименьшее положительное представимое число ε такое, что $1.0 + \varepsilon \neq 1.0$ в заданном формате.

Точность выполнения операций часто измеряется в единицах, равных весу самого младшего разряда мантиссы результата, называемых ULP (*unit in the last place*). Например, расстояние между 1.0 и $(1.0 + \varepsilon)$ составляет как раз 1 ULP. При удвоении числа ULP также удваивается.

При выполнении арифметических операций, а также вычислении квадратного корня (`sqrt`) результат округляется к ближайшему представимому в текущем формате числу, т. е. имеет точность 0.5 ULP. Одним из следствий такого округления является точный результат `sqrt` от квадрата целого числа (если он представим). Вычисление тригонометрических функций, степеней и логарифмов обычно выполняется не столь точно, давая погрешность в 1–5 ULP и более.

Благодаря двоичному формату чисел IEEE-754, при интерпретации их как целых[2], их разность в случае совпадения знака численно равна расстоянию между ними в ULP меньшего из них по величине.

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.

```
bool absEqual(double a, double b, double eps)
{ return abs(a - b) <= eps; }
```

Однако для сравнения произвольных чисел подобрать фиксированное значение константы `eps` часто невозможно. В этом случае следует сравнивать числа по относительной величине погрешности.

```
bool relEqual(double a, double b, double eps)
{ return abs(a - b) <= eps*max(abs(a), abs(b));}
```

[2]В редких случаях может потребоваться обращение порядка байт.

В качестве `eps` можно брать кратное `DBL_EPSILON`.

```
bool relEpsilonEqual
    (double a, double b, double epsilons)
{ return abs(a - b) <= (DBL_EPSILON * epsilons)
                     * max(abs(a), abs(b)); }
```

Наконец, можно оценивать погрешность непосредственно в ULP. В отличие от предыдущих вариантов, `ulpEqual` вернет `true` для бесконечностей одного знака. В то же время расстояние в ULP от `DBL_MAX` до $+\infty$ равно единице. Нечисла также могут оказаться «почти равными». Напротив, расстояние между -0 и $+0$ получается очень большим.

```
#include <cstdint> // int64_t
bool ulpEqual(double a, double b, int ulps) {
  if (a == b) return true;
  union {
    int64_t i; // в дополнительном коде
    double f;  // в формате binary64
  } x, y;
  x.f = a;
  y.f = b;

  // разные знаки?
  if ((x.i ^ y.i) < 0) return false;
  // разность в ULP
  return abs(x.i - y.i) <= ulps;
}
```

Предположим, что $a \geqslant +0$ (поведение отрицательных чисел аналогично), тогда при увеличении a, интерпретируемого как целое, на единицу получаем:

- следующее представимое число $b = (1.0 + \varepsilon)a$;
- либо $+\infty$, если a — наибольшее нормализованное число;
- либо нечисло, если $a = +\infty$.

Простым циклом можно перечислить все представимые неотрицательные числа с плавающей точкой:

```
void printFloats() {
  union { int32_t i; float f; } x;
  for (x.i = 0; x.i <= 0x7F800000; x.i++)
    cout << x.f << '\n';
}
```

Далее приведены два примера, демонстрирующие важность понимания принципов работы с числами с плавающей запятой.

Параметр линейной интерполяции

Пусть некоторая функция времени задана в конечном числе узлов и требуется ее приблизить ломаной. Чтобы получить параметр линейной интерполяции на отрезке ломаной, требуется преобразовать текущее время. Это можно попробовать сделать следующим образом:

```
// t — время, t1 — следующий узел, h — шаг
float lerpParamBad(float t, float t1, float h) {
  const float t0 = t1 - h; // предыдущий узел
  return (t - t0) / h;
}
```

Однако округление при вычитании существенно различных по величине `t1` и `h` может приводить к выходу конечного результата за пределы $[0, 1]$. Исправленный вариант может выглядеть так:

```
float lerpParam(float t, float t1, float h) {
  const float dt = t1 - t; // сколько осталось
  return (h - dt) / h;
}
```

Суммирование

Предположим, требуется вычислить арифметическое среднее большого набора чисел. Простое последовательное суммирование может приводить к очень большой погрешности.

Например, последовательное суммирование миллиарда единиц в формате binary32 даст 2^{24} (*почему?*).

Алгоритм, называемый *суммированием Кэхэна* или *компенсационным суммированием*, позволяет получить гораздо более точный результат, суммируя отброшенные при округлении основной суммы части в отдельной переменной — *компенсации* `comp`. Относительная погрешность этого алгоритма ограничена сверху значением[3]

$$(2\varepsilon + O(n)\varepsilon^2)\frac{\sum_{i=0}^{n-1}|x_i|}{|\sum_{i=0}^{n-1}x_i|}.$$

```
float sumKahan(float x[], size_t n) {
  float sum = 0.0f, comp = 0.0f;
  for (size_t i = 0; i < n; ++i) {
    const float y = x[i] - comp, t = sum + y;
    comp = (t - sum) - y;
    sum = t;
  }
  return sum;
}
```

[3]См.: *Higham N.* The accuracy of floating point summation // SIAM J. Sci. Comput. 1993. Vol. 14, No. 4. P. 783–799.

Справочный материал по cmath

функция	аргумент	значения
`sin`(x)	$\mathbb{R}$	$[-1, 1]$
`cos`(x)	$\mathbb{R}$	$[-1, 1]$
`tan`(x)	$x \neq (2k+1)\frac{\pi}{2}$, $k \in \mathbb{Z}$	$\mathbb{R}$
`asin`(x)	$[-1, 1]$	$[-\frac{\pi}{2}, \frac{\pi}{2}]$
`acos`(x)	$[-1, 1]$	$[0, \pi]$
`atan`(x)	$\mathbb{R}$	$[-\frac{\pi}{2}, \frac{\pi}{2}]$
`atan2`$(y,\ x)$	$\mathbb{R}^2$	$[-\pi, \pi]$
`sinh`(x)	$\mathbb{R}$	$\mathbb{R}$
`cosh`(x)	$\mathbb{R}$	$[1, +\infty)$
`tanh`(x)	$\mathbb{R}$	$[-1, 1]$
`exp`(x)	$\mathbb{R}$	$(0, +\infty)$
`log`(x)	$(0, +\infty)$	$\mathbb{R}$
`log10`(x)	$(0, +\infty)$	$\mathbb{R}$
`abs`(x)	$\mathbb{R}$	$[0, +\infty)$
`sqrt`(x)	$[0, +\infty)$	$[0, +\infty)$
`hypot`$(x,\ y)$	$\mathbb{R}^2$	$[0, +\infty)$
`pow`$(x,\ y)$	$(0, +\infty) \times \mathbb{R} \cup (-\infty, 0) \times \mathbb{Z} \cup \{0\} \times (0, +\infty)$	$[0, +\infty)$
`fmod`$(x,\ y)$	$y \neq 0$	$[0, \lvert y\rvert \operatorname{sgn} x)$
`ceil`(x)	$\mathbb{R}$	$\mathbb{Z}$
`floor`(x)	$\mathbb{R}$	$\mathbb{Z}$
`ldexp`$(x,\ n)$	$\mathbb{R} \times \mathbb{Z}$	$\mathbb{R}$
`modf`$(x,$ `n`$)$	$\mathbb{R}$	$(-1, 1)$

Примечания

- Все углы задаются в радианах.
- Вещественные числа приближаются некоторым конечным набором рациональных чисел в соответствии с возможностями встроенных типов `float` и `double`.

- `atan2(`y`, `x`)`: арктангенс $\frac{y}{x}$ в расширенном диапазоне, применяется в случае, когда требуется определить угол между осью $0x$ и отрезком $(0,0) - (x,y)$.

- `hypot(`x`, `y`)`: гипотенуза прямоугольного треугольника с катетами x и y. Вычисляется обычно по формуле $u\sqrt{1+(v/u)^2}$, где $u = \max(|x|,|y|)$, $v = \min(|x|,|y|)$.

- `pow(`x`, `y`)`: x^y, возведение в степень, для нецелых y вычисляется как `exp(`y` * log(`x`))`.

- `ceil(`x`)`: $\lceil x \rceil$, наименьшее целое, не меньшее x.

- `floor(`x`)`: $\lfloor x \rfloor$, наибольшее целое, не большее x.

- `fmod(`x`, `y`)`: возвращает $x - ny$ с таким целым n, что результат имеет знак x и по модулю меньше $|y|$.

- `modf(`x`, n)`: возвращает дробную часть числа x, целую часть записывает по адресу `n`, знак обеих частей совпадает со знаком x. Для извлечения экспоненты (по адресу `e`) можно использовать вызов `frexp(`x`, e)`.

- `ldexp(`x`, `n`)`: возвращает $x \cdot 2^n$.

Учебное издание

Кувшинов Дмитрий Рустамович

КОМПЬЮТЕРНЫЕ НАУКИ
Основы программирования

Учебное пособие

Заведующий редакцией	М. А. Овечкина
Редактор	Н. В. Шевченко
Корректор	Н. В. Шевченко
Оригинал-макет	Д. Р. Кувшинов

План выпуска 2015 г. Подписано в печать 03.03.2015.
Формат $60\times84^{1}/_{16}$. Бумага офсетная. Гарнитура Times.
Уч.-изд. л. 5,0. Усл. печ. л. 6,0. Тираж 70 экз. Заказ 127.
Издательство Уральского университета
620000, г. Екатеринбург, пр. Ленина, 51.
Отпечатано в Издательско-полиграфическом центре УрФУ
620000, Екатеринбург, ул. Тургенева, 4.
Тел.: +7 (343) 350-56-64, 350-90-13
Факс +7 (343) 358-93-06
E-mail: press.urfu@mail.ru

Для заметок